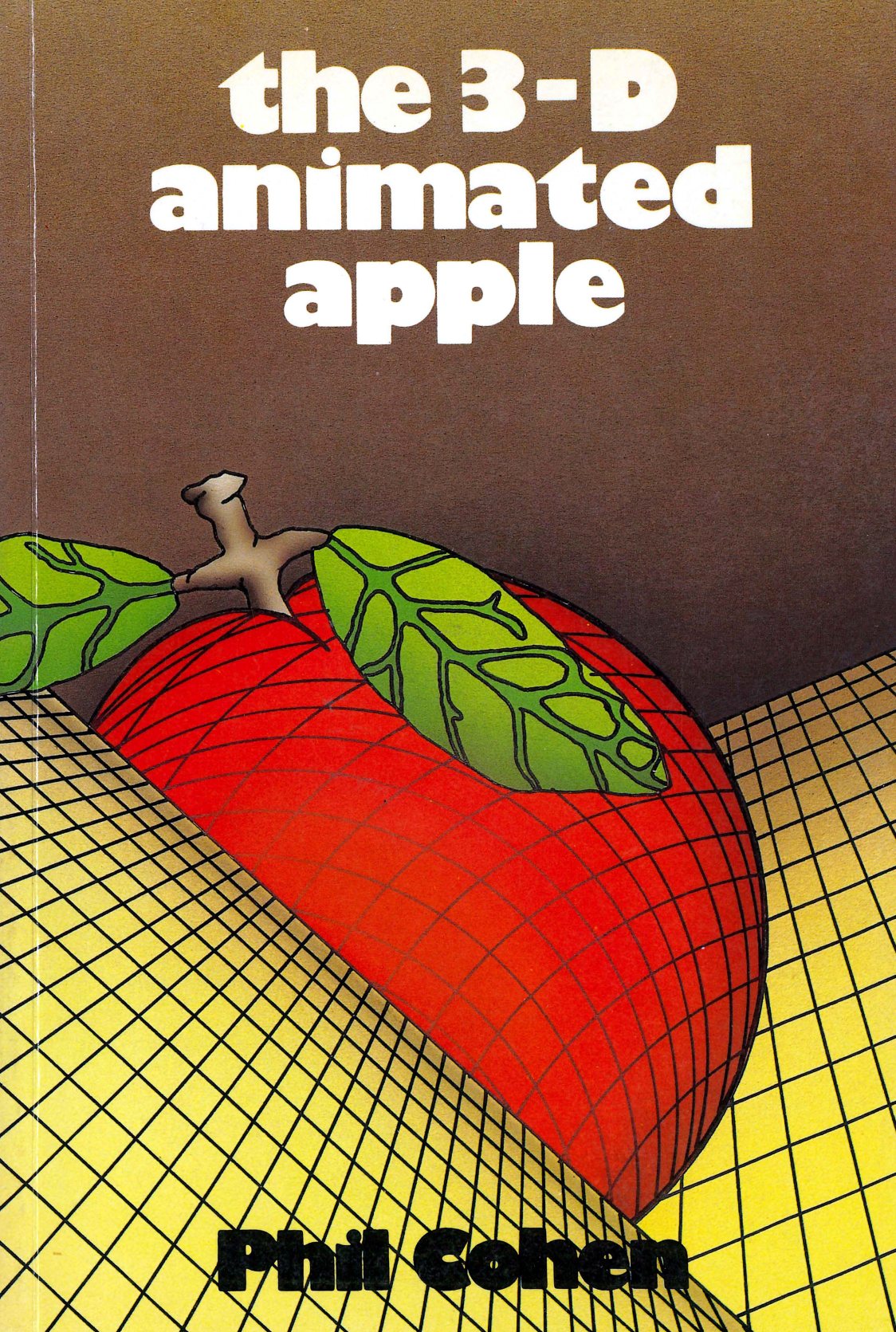
the 3-D animated apple



Phil Cohen

The 3-D Animated Apple

The 3-D Animated Apple

PHIL COHEN

PRENTICE-HALL OF AUSTRALIA

© 1983 by Prentice-Hall of Australia Pty Ltd.
All rights reserved. No part of this book may be reproduced in any form without permission in writing from the publishers. The program material contained herein or in any further deletions, addenda, or corrigenda to this manual or associated manuals or software is supplied without representation or guarantee of any kind. These computer programs have been developed for student use in a teaching situation and neither the authors, nor Prentice-Hall Inc., assumes any responsibility and shall have no liability, consequential or otherwise, of any kind arising from the use of these programs or part thereof.

Prentice Hall of Australia Pty Ltd, Sydney
Prentice-Hall International Inc, London
Editora Prentice-Hall do Brasil Ltda., Rio de Janeiro
Prentice-Hall of Canada Inc, Toronto
Prentice-Hall of India Private Ltd, New Delhi
Prentice-Hall of Japan Inc, Tokyo
Prentice-Hall of Southeast Asia Pte Ltd, Singapore
Whitehall Books Ltd, Wellington
Prentice-Hall Inc, Englewood Cliffs, New Jersey

ISBN 0-13-920224-2

Typeset by Abb-typesetting,
Collingwood, Victoria.
Printed and bound in Australia by
Globe Press Pty Ltd, Brunswick, Victoria.

1 2 3 4 5 87 86 85 84 83

Apple and Apple II are registered trademarks of Apple Computer, Inc.

Library of Congress Cataloging in Publication Data

Cohen, Philip Jonathan, 1956–
The 3-D animated Apple.

1. Computer graphics. 2. Computer animation.
3. Apple II (Computer) — Programming. I. Title.
II. Title: The three-D animated Apple.

T385.C543 1983 001.64'43 83-11141
ISBN 0-13-920224-2

National Library of Australia
Cataloguing-in-publication data

Cohen, Phil, 1956–.
The 3-D Animated Apple.

ISBN 0 7248 1293 8.

1. Apple computer — Programming. 2. Computer graphics. I. Title.

001.64'24

Contents

Introduction vii

1 Shapes 1

2 Figures in Two Dimensions 55

3 Three Dimensions 119

Appendix 1 Binary 185

Appendix 2 Attaching a Printer 193

To my Father

Introduction

In recent years, computers have invaded (excuse the pun) more and more aspects of our day-to-day lives. Some of the more exciting applications of computers come from their ability to handle pictures. Computers are becoming more capable of displaying and altering images at our command, and this gives us a powerful tool for communication.

Computer graphics have many, many applications. The most apparent is in the myriad computer games that have been developed in recent years. However, more challenging applications are emerging in medicine, the military, engineering and in many other fields where precision and clarity are required. The commercial arts and, in some cases, the creative arts are also feeling the impact of computer graphics.

This book is intended to lead anyone with an interest in computing through some of the programming techniques that will allow an Apple II computer to handle pictures. It is not written for a specific application. In fact, it is deliberately written to provide a basis from which almost any application can grow.

What you will need

To run the programs in this book, you will need an Apple II computer with at least 48K of memory and a television or monitor interfaced to it. A cassette recorder or a disc drive would make it possible to record the programs and avoid having to retype them each time you want to use them. If you are at all serious about using the programs in this book for animation or other uses, then I suggest that a disc drive is an essential item of equipment.

A disc containing all the programs listed in this book can be purchased from:

Prentice-Hall of Australia
7 Grosvenor Place
Brookvale, New South Wales
Australia 2100

The price within Australia is A$12.00. The price for orders from all other countries is $US12.00 (includes handling and air mail postage).

In time you may want to buy a printer. See Appendix 2 for information on this. The printer is optional however. All you need to get started is an Apple.

Shapes

Chapter 1

The Apple is probably one of the best microcomputers on the market for graphics and animation. As well as being able to draw lines from point to point the Apple can be 'taught' whole shapes. A shape is 'taught' to the machine by storing it in the Apple's memory. After this has been done, the shape (or shapes — any number can be stored) can be displayed on the screen using commands from the Apple's BASIC language.

There are BASIC commands which can be used within programs to manipulate stored shapes in various ways: shapes can be re-drawn at different positions on the screen; they can be reduced or enlarged; and they can be rotated. These changes can be made quickly enough so that the shapes appear to be moving about on the screen.

These features make the Apple II very suitable for playing games that involve fast-moving pictures.

Your Apple will have at least 48K of memory — that means that it has the capacity to store $48 \times 1024 = 49152$

eight-bit numbers. So there are 49152 'locations' in the Apple's memory. Each location is given a number from 0 to 49151. There are two BASIC commands that allow you to look at and alter the contents of each location: PEEK() and POKE. PEEK() lets you 'see' what is in each location, e.g. 'PRINT PEEK(30000)' will print the value that's held in location 30000. POKE lets you *alter* the contents of a particular location, e.g. 'POKE 30000, 15' will set location 30000 to 15 (00001111 in binary; see Appendix 1).

The commands PEEK and POKE are used to 'teach' the Apple the shapes to be drawn onto the screen. The way this is done is to 'POKE' a series of numbers which represent the shape into the machine's memory directly. When it is in memory, this series of numbers is called a 'shape table'. The Apple will usually only have one shape table in it at any given time, but the shape table can hold any number of shapes.

I should stress at this point that, no matter *what* you do with the POKE and PEEK commands it is *not* possible to damage the computer in any way, so do not be afraid to use them. The worst that you could do would be to temporarily 'hang' the machine, by overwriting some of the memory used by BASIC. What happens then is that the Apple will stop what it's doing, and may put some strange things onto the screen. Don't panic — there is *nothing* you can do to harm it. The first thing to try is pressing RESET. If that doesn't work, then simply turn the machine off, wait a few seconds, and then turn it back on.

Before we can store a shape table in memory we must find a suitable area in memory where it can be stored safely. If we put it in a memory area that has been used already it will 'overwrite' what was there before. The

previous contents of that area, which might have been programs or part of the screen display, would be lost.

The Apple's BASIC will use all of the memory available to it (if you put in a big enough program), so there has to be some way of 'reserving' a part of memory to put the shape table in. The BASIC command HIMEM: can be used to do just that.

How to use HIMEM:

HIMEM: sets the highest memory location that BASIC will use. To use HIMEM:, you simply have to include a line such as 'HIMEM: 30000' in your program. In this case BASIC would only use memory locations *up to* 30000 for programs, variables, etc. Anything that is stored in the memory locations *above* 30000 will be quite safe — BASIC will not overwrite these locations. So we can create a safe area in the high part of memory where we can put our shape table, and set HIMEM: to just *below* it. In practice we find out what HIMEM: *is* (i.e. what the highest available memory location is), move it down to make room for the shape table, then put the shape table in the memory area *above* HIMEM:.

How do we find out what HIMEM: is set at? There is no BASIC command that will tell us. The value of HIMEM: is stored in memory locations 115 and 116. Two locations are needed to store the value of HIMEM because each location of the Apple's memory can only store an eight-bit binary number (that means a base ten number from 0 to 255). The value of HIMEM: can be any number

between about 2051 (because BASIC occupies some memory even when there is no program) and 49151 — the top of memory. Location 116 holds the number of times 256 will divide into the value of HIMEM:, and location 115 holds the remainder. So HIMEM: can be found by using 'PRINT PEEK(116)*256+PEEK(115)'. Note that the *first* location (115) is the one that stores the 'remainder' part. This is the usual method of storing 'addresses' (i.e. location numbers) in the Apple II using two successive locations.

We can now find HIMEM:, store a shape table in memory, and protect it by setting HIMEM: past the bottom of the table. That gives us a shape table in memory, all ready for us to use from BASIC. In order to use the shape table in our programs, we must be able to tell the computer where the table starts. The address of the beginning of the shape table is stored (in exactly the same way as HIMEM:) in memory locations 232 and 233 — again, location 232 holds the 'remainder' part, and 233 the 'number of 256s'. We can find out the number stored in these locations using the command 'PRINT PEEK(233)*256+PEEK(232)'.

When the computer is first turned on, the start of the shape table is set automatically to 65535 — in other words, both 232 and 233 are set to 255. This is to 'remind' the machine that there is *no* shape table in it yet. So as well as moving HIMEM: and putting the shape table into memory, we will have to POKE locations 232 and 233 to 'point' to the start of the shape table. Although we can change the value of HIMEM: quite simply, e.g. 'HIMEM: 30000', to set the starting address of a shape table we have to POKE each of the two locations (232 and 233) separately.

Shape table

The shape table[1] is made up of three parts: the first part (in the first two locations of the table) tells the computer how *many* shapes are in the table; the second part (which starts at the third location of the table and uses two locations per shape) tells the computer *where* each of the shapes in the table starts; the third part contains the shapes themselves as a series of numbers — each shape ends with a value of 0 in the last location.

Figure 1-1 shows how the shape table is set out in memory.

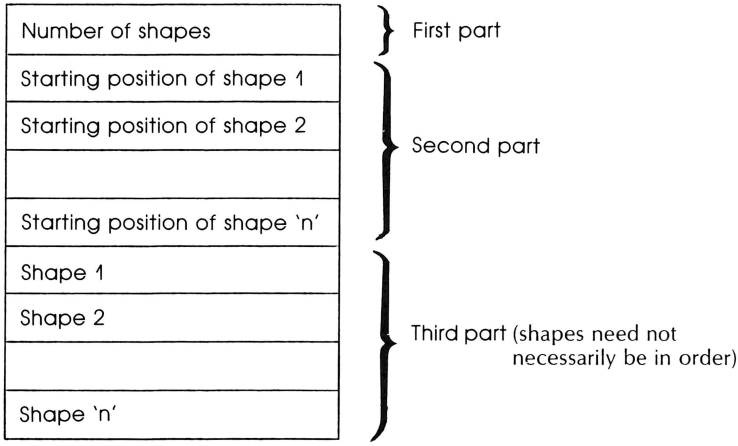


Figure 1-1

[1] At the end of this chapter, I've included two programs. The first allows you to make up a single shape automatically, and the second allows you to combine several shapes into a shape table, to store it in memory (so that you can try the shapes out), and to produce automatically a program which will POKE that table into memory. So if you're trying to make up a shape table *now*, don't use the procedures given here directly — use the programs instead.

Let's have a look at a typical shape table.

Position in table		*Value*	*Comments*
0	First part	2	number of shapes
1		0	
2	Second part	11	starting position of shape 1
3		0	
4		6	starting position of shape 2
5		0	
6	Third part	36	shape 2
7		45	shape 2
8		54	shape 2
9		63	shape 2
10		0	end of shape 2
11		100	shape 1
12		54	shape 1
13		3	shape 1
14		0	end of shape 1

The first location in the table (location 0) holds the value 2— the number of shapes in the table. The next location holds zero. Then there are two sets of two locations, which tell the computer where the two shapes start (in relation to the start of the table). Generally, only the first location of each pair will be used. The second of each pair is the number of 256s — as in the storage of HIMEM:, etc.

So the first shape (shape number 1, when you come to use it in BASIC) starts 11 locations from the start of the table, and the second shape starts 6 locations from the start of the table. Notice that both shapes end with a zero which indicates the end of a shape. So shape 1 runs from position 11 in the table to position 14, and shape 2 runs from position 6 to position 10.

What the computer does with each entry in the shape table is to interpret it as three groups of binary digits. For example, let's look at the shape that starts at position 11 in the table. Its first entry is 100. If we convert this number to binary, we get 01100100. The computer, when it comes to draw the shape, will see this binary number as two groups of three digits and one of two digits, i.e. 01 100 100. Having split it up that way, the computer will add a zero to the first group, giving three groups of three digits, i.e. 001 100 100.

All the rest of the entries in the table will be split up in this way so that each location will hold three groups of three binary digits (and the first group will always start with a zero).

The computer will interpret each of those three-digit groups in the following way: the first digit in the group will tell the computer whether to draw something, or just to move the point that it's drawing from; and the next two digits will tell the computer which direction to move or draw in.

The right-most group of three digits is done first, so that in the above example the group 100 would be the first to be done. This would be interpreted as 'draw a line on the screen' (first bit = 1) and 'draw upwards' (second/third bits = 00). The direction is determined by the value of the last two bits as follows:

Value	*Direction*
00	Up
01	Right
10	Down
11	Left

The first entry in shape 1 which is 100 (in decimal) will be converted by the computer into the binary number 001 100 100 and processed from right to left as follows:

Value	*Meaning*
100	Draw a line up one step
100	Draw a line up one step
001	Move (without drawing) right one step

So the first entry, or 'byte', of shape 1 is telling the computer to draw a line two steps long upwards, and then to move one step to the right at the top of that line.

If we convert all the entries in shape 1 into binary we get the following:

Entry	*Groups*	*Groups that will be processed*
100	001 100 100	001 100 100
54	000 110 110	110 110
3	000 000 011	011

The computer will ignore *any* group that is all zeros, and which has no bits on to the left of it. So the left-most group of the '54' entry will be ignored, and the two left-most groups of the '3' entry will be ignored, too.

If the last entry had been 131 (i.e. 128 + 3), then the groups for that entry would have been 010 000 011 — none of these three groups would have been ignored. But because there is nothing to the left of the middle group in

the '3' entry, then *both* of the two left-most groups are ignored.

Starting at the top and going from right to left for each entry, the groups will be processed in the following order: 100 100 001 110 110 011.

The computer will interpret these groups as follows:

Group	*Meaning*
100	Draw up
100	Draw up
001	Move right
110	Draw down
110	Draw down
011	Move left

This part of the table draws two parallel lines, each two steps long, and returns the 'pointer' to the initial starting point.

Trying it out

Press the RESET key on your Apple, and type in the following program (or get it off disc if you have one[2]):

[2] A disc containing all the programs in this book is available — see the introduction.

PROG 1

```
10  HIMEM: 38383
20  POKE 232,240: POKE 233,149
30  FOR I = 1 TO 15
40  READ N
50  POKE 38383 + I,N
60  NEXT I
70  DATA  2,0,11,0,6,0,36,45,54,63
80  DATA  0,100,54,3,0
```

When you run this program it will 'POKE' the shape table into the right area in the Apple's memory, and will set HIMEM: and the starting point of the table to the right places.

The following commands (type them in) will put the Apple into graphics mode.

HGR	sets the screen to 'high resolution graphics' mode with four lines of text at the bottom. Type in 'TEXT' to return the screen to normal mode later.
HCOLOR=3	selects a color to draw in.
ROT=0	determines the rotation of the shape.
SCALE=5	determines the size of the shape.

Now we're ready to draw! First, draw the two parallel lines by typing:

```
DRAW 1 AT 40, 50
```

This instruction tells the computer to draw shape 1 from the shape table starting at a position on the screen given by the numbers 40, 50.

When the Apple is in graphics mode, the screen can be thought of as a piece of fine-grid graph paper: each of the imaginary squares can be colored in or left blank, and by numbering all the rows and columns it is possible to pin-point any position on the screen. The Apple's graphics system divides the TV screen into 280 columns (the left-most column is numbered 0, the right-most 279) and 160 rows (the top row is 0, the bottom is 159 (when in HGR)).

So, the command 'DRAW 1 AT 40, 50' tells the computer to start drawing at grid reference 40,50 — that is the 40th column out from the left of the screen and the 50th row down from the top of the screen. The first number always gives the distance from the left of the screen and the second gives the distance down from the top of the screen. So, the top left corner is numbered 0,0, while the bottom right-hand corner is numbered 279,159.

Let's look at shape 2 which is also stored in memory. Type in this line:

```
DRAW 2 AT 70, 70
```

A box should appear. Now try 'decoding' the second shape's entries in the table to find out *how* the box was drawn.

The DRAW command tells the Apple to put onto the screen the shape whose number is given, at a position

given by the pair of numbers after 'AT'. This will be the *starting* position of the shape. The parallel lines we drew earlier will have their *bottom corner* (the place where the drawing started from) at the position given in the DRAW command, i.e. 40, 50.

You will notice that both of the shapes given in the above example finish at the same point that they started on: this is not essential, but is usually a good idea because of what happens if you type something like: 'DRAW 1'. In other words, what happens if you miss out the position numbers? Then the Apple will start drawing from the place where the last shape ended. Supposing you just drew a shape that started at, say, 50, 50 and ended up at 55, 65; if you use DRAW without the position numbers, you will draw a shape starting at 55, 65 — the new position.

By designing all your shapes so that they end up at the same point that they started on, you can use DRAW alone to redraw the shape at the same place — and that can be useful, as you'll find out later.

One thing to be aware of is that the Apple won't draw *anything* unless you use AT in your *first* DRAW statement after you type in HGR. That's because it won't know where to start.

Color

The command HCOLOR sets the color of the lines that are drawn on the screen. HCOLOR=3, as used earlier, sets the color of all lines drawn after that to color 3 — that

is, white. The colors are allocated numbers according to this table:

HCOLOR	*Color*
0	black
1	green (depends on TV set)
2	blue (depends on TV set)
3	white
4	black
5	? (depends on TV set)
6	? (depends on TV set)
7	white

(If you are using a black-and-white TV or monitor, it is best to use colors 0 and 3.)

You can change color anywhere in a program (by using HCOLOR=), and everything drawn from then on will be in the new color.

The command SCALE alters the *size* of the shape that's drawn on the screen. Shapes drawn with SCALE=1 will be rather small — each 'step' in the shape table will correspond to one 'step' on the screen. The SCALE used earlier (in PROG 1) was 5 — each 'step' in the shape table corresponded to 5 steps on the screen. Using a SCALE of less than 1 doesn't work. You can change SCALE anywhere in the program.

ROT is one of the most powerful commands that the Apple has for graphics. What it does is to *rotate* the shape about its starting point. ROT=8 will rotate the shape 45 degrees clockwise. ROT can be altered at any point in the program. Any value from ROT=0 (which will not rotate the shape at all) to ROT=64 (which will rotate it *all* the way round, and leave it back where it started) can be used.

Having drawn a shape somewhere on the screen, suppose you want to erase it? You *could* just change HCOLOR=0 (which gives black), and then do the DRAW again. That would go round the same shape, drawing it in in black. This would make the shape 'disappear'. However, this way of erasing has one disadvantage — if the shape is 'on top' of something which is already on the screen (a background, for example), then you will be drawing black lines all over the background. We can avoid this problem by using the *XDRAW* command, which draws the shape in exactly the same way as the DRAW command, but using the color *opposite to the one that is in the background*. This means that if the place that's about to be drawn on is black, then the computer will draw white there. (If it's blue, it will draw green, and so on.)

If you XDRAW a shape over an existing background, the computer will, along the line of the shape, make each of the points on the screen the *opposite* color to what they were. The advantage of this is that if you XDRAW a shape and then XDRAW it again in the same place, it will disappear, and the background will be left as before!

The ability to erase shapes quickly can be very useful. Suppose we drew those parallel lines on the screen, then erased them, and quickly re-drew them a little to the right? If it were fast enough, it would look as if they had *moved* from one spot to the other. And if we repeated it, it would look as if the motion was continuous.

Modify the program we used earlier (to put the parallel lines and square into memory as a shape table) by adding lines 90 to 180 as shown in PROG 2.

PROG 2

```
10  HIMEM: 38383
20  POKE 232,240: POKE 233,149
30  FOR I = 1 TO 15
40  READ N
50  POKE 38383 + I,N
60  NEXT I
70  DATA  2,0,11,0,6,0,36,45,54,63
80  DATA  0,100,54,3,0
90  HGR
100  ROT= 0
110  SCALE= 5
120 Y = 50
130  FOR X = 10 TO 200
140  XDRAW 1 AT X,Y
150  FOR I = 1 TO 10: NEXT I
160  XDRAW 1
170  NEXT X
180  TEXT
```

Line 90 is the HGR command, which sets the screen so that the top 4/5ths of it is for high-resolution graphics, and the bottom four lines are for text.

Line 100 sets the rotation to zero (i.e. no rotation for the shapes).

Line 110 sets the scale to 5, which gives an appropriately sized shape on the screen (with the parallel lines, whose lines are two steps long in the table, it will give a figure which is 10 screen steps long).

Line 120 sets variable Y to 50. We're going to use variables X and Y for the position across and down the screen, respectively. So setting Y to 50 means that the drawing is going to be done about one-third of the way down the screen (as the bottom of the HGR screen will be at Y=159).

The loop at lines 130 and 170 gives X (which, remember, is the distance from the left of the screen) the values 10, 11, 12 . . . 198, 199, 200. So the position across the screen is going to change from near the left of the screen to near the right of the screen in small steps.

Line 140 XDRAWs the shape (in this case, shape 1, which is the parallel lines) at the position given by X and Y.

Line 150 looks curious — but all it does is to slow the program down a little. This may seem like a funny thing to do to a computer program, but its usefulness will become obvious later.

Line 160 XDRAWs the shape again (at the *same* position), making the shape *disappear* from where it was, and momentarily leaving the screen blank.

Line 170 sends the program back round in the loop, so that X will be increased by 1 again, and the program will go back to line 130.

The first time through the loop, the parallel lines will be drawn at X=20 (i.e. at the left of the screen). Then there is a short delay (given by line 150), and then the parallel lines are erased, X is increased by 1, and they are re-drawn at X=21. As this process is repeated (draw, delay, erase, move, draw, delay, erase, move, . . .) it will seem as if the parallel lines are actually moving smoothly from one side of the screen to the other. The program takes a couple of seconds to run, during which the parallel lines seem to start at the left of the screen and move slowly across to the right.

The last line (line 180) merely sets the screen back into its normal mode — with the whole of the screen given over to text.

By changing the values of X and Y in different ways as the program progresses, you can change the path of the parallel lines. Try adding this line to the program:

```
135 Y=50+30*SIN(X/8)
```

The parallel lines will seem to 'spiral' from one end of the screen to the other!

Lunar Lander

Let's now try using all these tricks in a game: the lunar lander program. This class of games programs grew out of the American Apollo missions — it was intended to imitate the 'simulators' used by the astronauts while training to land the craft on the surface of the Moon. The game also has the interesting property of being almost totally addictive.

The idea is that the craft is falling under lunar gravity, and you have to adjust the thrust from the rockets so that it lands at a reasonable speed. That may seem easy, but the problem is that fuel is limited. The player must use fuel at such a rate that the craft will land slowly, but will not run out of fuel before the landing is completed. If the craft runs out of fuel, it will crash on the lunar surface.

In order to make the game more interesting, it is traditional to show a small representation of the craft as it

comes down towards the surface. The program also displays the current downward speed (and whether it's too high for landing) and the remaining fuel on the 'text' area at the foot of the screen.

This is how the game works: when you RUN the program, current fuel and speed will show at the foot of the screen. The craft will appear at the top left of the screen and will start to slowly fall towards the bottom, and move to the right. You adjust the thrust by pressing the number keys — 0 will give no thrust, 9 will give maximum thrust, and 5 will *just* balance the acceleration due to gravity, and will keep the speed constant.

The craft falls against a background of horizontal lines. These are intended partly as a guide to its position, and partly to show that by using XDRAW we can move an object without disturbing the background.

If the craft lands too fast, the program uses the ROT command to draw the shape of the craft on its side on the surface — a crash landing!

The program uses a shape table made up using the programs MAKE SHAPE and COMBINE SHAPES, given at the end of this chapter. These programs enable you to make up individual shapes and to combine them into shape tables. Shape number 1 in the table is the craft itself:

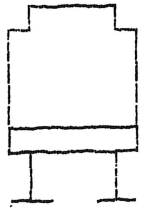


Figure 1-2

Shapes 2 and 3 are two parts of the craft's 'rocket exhaust':

Figure 1–3

Figure 1–4

By quickly drawing first shape 2 and then shape 3, we provide the illusion of a 'flickering' exhaust, like a candle flame. When both are drawn it looks like this:

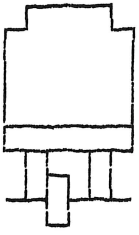


Figure 1–5

When the program is in operation the screen display looks something like Figure 1–6.

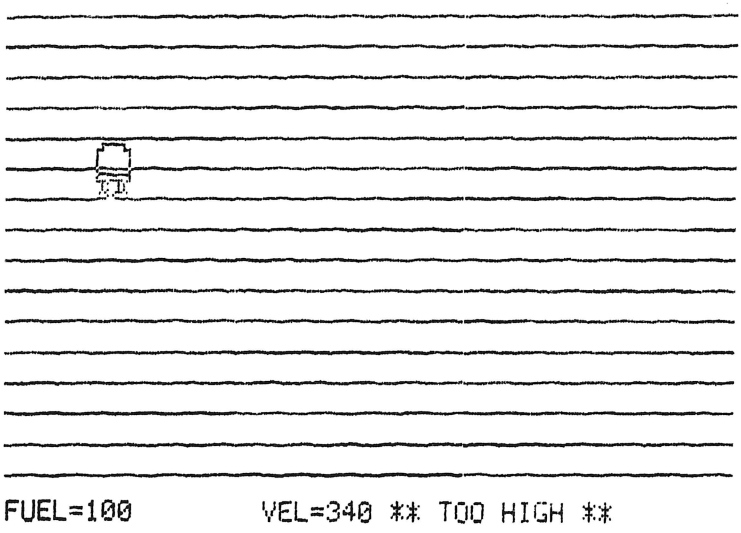


Figure 1-6

LANDER

```
10  GOSUB 4000
20 X = 5:Y = 0
30  REM  X HORIZ, Y VERT
40 VS = 0
50 A = 10
60 T = 0
70 F = 100
80 MV = 20
90  REM  START OF LOOP
100  XDRAW 1 AT X,Y
110  XDRAW 2
120 N =  PEEK ( - 16384)
130  IF N > 185 OR N < 176 THEN
GOTO 150
140 T = N - 176
```

```
150 X = X + 1
160  IF X > 279 THEN  PRINT : PRINT :
PRINT "** OVERSHOT **": STOP
170 F = F - T / 10
180  IF F < 0 THEN F = 0:T = 0
190 VS = VS + A - T * 2
200 Y = Y + VS / 100
210  IF Y > 155 THEN  GOTO 320
220  XDRAW 3
230  HOME : VTAB 21
240  PRINT "FUEL="; INT (F),
"SPEED=";VS;
250  IF VS > MV THEN  PRINT " ** TOO
HIGH **"
260  XDRAW 2
270  XDRAW 3
280  XDRAW 1
290  REM  END OF LOOP
300  GOTO 90
310  REM  NOW FOR THE EXIT POINTS ...
320  IF VS > MV THEN  GOTO 350
330  PRINT : PRINT "** SUCCESSFUL
LANDING! **"
340  STOP
350  XDRAW 1: XDRAW 2
360  ROT= 8
370  XDRAW 1
380  PRINT "** CRASHED! **"
390  STOP
4000  HGR
4010  SCALE= 2
4020  HCOLOR= 3
4030  ROT= 0
4040  FOR I = 9 TO 159 STEP 10
4050  HPLOT 0,I TO 279,I
4060  NEXT I
5000  HIMEM: 38354
5010  POKE 232,211: POKE 233,149
5020  FOR I = 1 TO 44
5030  READ N
5040  POKE 38354 + I,N
```

```
5050  NEXT I
5060 DATA 3,0,19,0,14,0,8,0,50,62
5070 DATA 36,69,3,0,51,53,37,28,0,63
5080 DATA 39,45,45,45,62,63,24,27,36,36
5090 DATA 37,45,45,46,54,182,51,46,59,27
5100 DATA 127,100,1,0,
5110  RETURN
```

The part of the program that actually puts the shape table into memory is in lines 5000–5110. These were generated using the MAKE SHAPE and COMBINE SHAPES programs at the end of this chapter. The programs produce a segment which starts at line 5000 — that leaves plenty of room for the rest of the program to go in before it.

Lines 4000–4060 are part of the start-up routine (which is called as a subroutine from line 10, right at the start of the LANDER program).

As soon as the LANDER program is run, line 10 causes the computer to do lines 4000–5100, and then line 5110 sends the computer back to line 20 for the rest of the program. So, lines 4000–5100 are the 'start-up' part of the program.

Line 4000 sets the machine into HGR mode, with high-resolution graphics in the top part of the screen, and four lines of text at the bottom.

Lines 4010–4030 set up the scale, color and rotation. Lines 4040–4060 look a little curious at first, but all they do is to draw horizontal lines across the screen at regular intervals — this is the 'background' for the LANDER program.

Line 5000 sets HIMEM: to a point just *below* the start of the shape table (the COMBINE SHAPES program works this out as it makes up the table).

Line 5010 sets up the pointer to the start of the shape table.

Lines 5020–5100 actually POKE the shape table into memory. Each time the program is run, these lines will make sure that the shape table is there. This method of storing shape tables has the advantage that the whole program (shape table and all) can be stored on tape, disc or whatever, as a single entity.

Going back to line 20 (as the computer would do, after processing line 5110), the program sets up the variables that we're going to use.

The variables X and Y are to be used to specify the position of the craft — so it starts at X=5 (very near the left of the screen) and Y=0 (at the top of the screen).

Line 40 sets the initial value of VS — the vertical speed of the craft. This starts at 0, and will increase due to gravity as the program proceeds (unless the rockets are used).

Line 50 sets A, the acceleration due to gravity. This has been made a variable so that it can be changed to make the game more difficult (or easier).

Line 60 sets T, the thrust — so that the program will start out with the rockets off (i.e. no thrust).

Line 70 sets F, the amount of fuel. When this reaches 0, the rockets will stop. It will decrease at a rate that depends on the thrust used.

Line 80 sets the maximum landing velocity — the highest safe landing speed. The program will 'crash' the craft if you land it at a speed greater than this. As the program is run, a message will appear if this speed is being exceeded.

The program operates in a 'loop' from line 90 to line 300. Each time it goes round the loop, it will update all of

the variables, and re-draw the craft on the screen. Lines 100 and 110 draw the craft on the screen, and also turn on one of the 'flame' shapes. Each time round the loop, the program will check to see if any of the keys has been pressed. Lines 120 and 130 do this, in a way that I won't go into. Suffice to say that when line 120 is reached, variable N will be set to the ASCII value of the last key pressed. Line 140 sets T (the thrust) to the actual value of the key (0 to 9).

Line 150 changes the horizontal position of the craft, moving it a little to the right, so that the craft appears to come down as if from orbit, adding realism to the game.

Line 160 is the exit point if you don't land the craft before it reaches the right-hand end of the screen.

Line 170 makes the adjustment to the amount of fuel (F) left. It will reduce by the thrust (T) divided by 10. There's no special reason for choosing 10 — it was found suitable by trial and error.

Line 180 determines what happens when we run out of fuel. The fuel is kept at 0 (to stop it going negative!), and the thrust is set back to 0 (no matter what you set it at from the keyboard). This ensures that the craft 'falls' to the surface.

Line 190 sets the vertical speed — it will be the current VS *plus* the acceleration due to gravity (A), *minus* the thrust (T). Again, the thrust is multiplied by a 'fiddle factor' of 2 to make the game realistic. Notice that with T=5 and A=10, the vertical speed will stay constant.

Line 200 works out the new position on the screen — it will be a function of how fast the craft is falling (VS). The 100 is a 'fiddle factor'.

Line 210 finds out when the craft reaches the surface. When this happens, the program jumps to line 320.

Line 220 turns on the second part of the flame.

Lines 230 and 240 print the current values of fuel and speed on the screen, and line 250 prints the message "✳✳ TOO HIGH ✳✳" if the speed is greater than the maximum safe landing speed (MV).

Line 260 turns *off* the part of the flame that we turned on at line 110 (remember, all of these shapes finish at the point where they started, and so there's no need to use DRAW . . . AT, as each successive use of DRAW will start at the same spot, which is set by X and Y in line 100).

Then lines 270 and 280 turn off the other part of the flame and the image of the craft too.

The sequence of using the shapes is as follows:

1) Shape 1 on (line 100)
2) Shape 2 on (line 110)
3) Shape 3 on (line 220)
4) Shape 2 off (line 260)
5) Shape 3 off (line 270)
6) Shape 1 off (line 280)

By the time we reach the end of the loop (at line 290), all of the shapes have been turned off, and the background is as it was at the start. When the loop is repeated, the program will draw the shapes again, but at the *new* position given by X and Y. So the whole thing will have moved.

The program will continue to go round the loop, moving the craft and updating the fuel and speed readings at the bottom of the screen. The craft will be re-drawn again and again, and so will appear to actually 'move' across and down the screen. This will go on until one of three things happen:

1. The craft reaches the right-hand edge of the screen — line 160 will print a message to this effect, and will stop the program.
2. The craft lands successfully — when the craft reaches Y=156, line 210 will send the program to line 320.
3. The craft crashes — again, when Y=156, line 210 will send the program to line 320.

At line 320, the program tests whether the landing speed (VS) is greater than the maximum safe landing speed (MV). If the speed is *below* the maximum, then lines 330 and 340 print a message and stop the program. However, if the speed is *above* the maximum (i.e. if the craft crashed), then lines 350 to 390 are done.

Line 350 will erase the images of the craft and of the first 'flame' shape (because, at this point in the program, they are still on).

Line 360 sets the rotation to 45 degrees clock-wise.

Line 370 draws the craft at *45 degrees rotation* at the place that it crashed. The effect is as if the landing had crushed one of the legs!

The 'crashed' craft looks like Figure 1–7.

Figure 1–7

The LANDER program is one of the classic games programs that use animated graphics. A version of LANDER is found on almost all mainframe (i.e. professional) computer installations capable of graphics.

MAKE SHAPE and COMBINE SHAPES

MAKE SHAPE allows you to 'draw' a shape onto the screen using the keyboard. It will then do all of the calculations necessary to generate a shape table entry. MAKE SHAPE allows you to store the results of these calculations in one of three ways (depending on what equipment you have): on disc; on tape; on paper (either by copying by hand from the screen, or using a printer).

Once you have a number of shapes that you wish to put into a shape table, the COMBINE SHAPES program will take the shape table entries for a number of shapes from any one of these sources, and combine them into a

single shape table. COMBINE SHAPES will then either leave the shape table set up in memory (so that you can test the shapes one by one in HGR mode), or will generate a series of statements (starting at line 5000) which can be made into a program *without retyping*.

Using MAKE SHAPE

It is a good idea to sketch the shape first on graph paper. Then it becomes a simple matter to put it into the computer.

When the program is RUN, the top part of the screen (the graphics part) will go blank except for a dot near the middle. The text part of the screen will show this message:

```
U=UP M=DOWN K=RIGHT H=LEFT J=ON/OFF
F=FINISH E=ERASE
```

The direction in which the drawing 'dot' moves is determined by pressing the appropriate key. If you press, say, the U key, a short line segment will appear, starting at the dot in the middle of the screen. By pressing the U, M, H and K keys, you can make the line segments follow the shape that you want to draw.

You can move without drawing by pressing the J key which turns the line 'on' and 'off'. If you press it once, you will find that the line on the screen is made up of dots; not

a solid line. It will stay like this until you press the J key again. The dotted segments of the line will not appear on the screen in the final shape — they are only showing you where you have 'moved' to (without drawing).

If you find that you have made a mistake, then you can use the E key to erase the last move. When you press the E key, the whole shape will disappear, and will be re-drawn with the last segment missing. You can then go on to complete the shape. When the shape is finished, you simply press the F key. This will cause the program to output the shape table entries.

If you are drawing a shape for use with the XDRAW command, it is worth remembering that for shapes where you want a line to 'double back' on itself (e.g. when drawing the top of a 'T', you would have to draw one 'branch', then backtrack and draw the other), you will have to turn the line 'off' when you are going over the previously drawn segment. If you 'double draw' a segment of a line, then XDRAW the shape, drawing over the same line segment again will cause it to disappear!

When you press the F key, you will get the message:

```
NOW MAKING UP THE TABLE
```

The binary values that make up the table entry will be displayed as the computer generates them. When that is finished, the program will ask you to select the form of output:

```
CHOOSE WHICH TYPE OF OUTPUT (S=SCREEN,
T=TAPE, D=DISC)?
```

Type S, T or D, depending on what equipment you have available. If you have a disc drive, the computer will put a copy of the CATALOG onto the screen for you, and then ask for a file name. The table will be put into this file (which will be a text file) for later use by the COMBINE SHAPES program. If you have a tape recorder attached to the Apple, pressing T will cause the computer to pause for you to set the recorder up, and then save the table entries using the STORE command, which puts an array onto tape. The tape can also be used with the COMBINE SHAPES program later.

If you have neither tape nor disc, you will have to type S. This will display the shape table entries on the screen (going from left to right across the top, then to the next row). If you have a printer, you could use it to record the entries, but if you do not, then you will have to copy them by hand. Fortunately, most shapes are fairly small, and so this will not take you long.

MAKE SHAPE

```
10  DIM A%(100,2)
20  HGR
25  VTAB 24
30  HCOLOR= 3
40  HPLOT 140,80
50 X = 140:Y = 80
60 P = 1: REM  NEXT AVAILABLE ELEMENT
OF A%(,)
70 O = 1: REM  1=ON, 0=OFF IN ARRAY
ELEMENT 2
80  PRINT
90  PRINT "U=UP M=DOWN K=RIGHT H=LEFT
J=ON/OFF"
```

```
100  PRINT "F=FINISH E=ERASE"
110  REM
120  REM  DRAW SHAPE ON THE SCREEN
130  GET A$
140  IF A$ < > "E" AND A$ < > "F"
AND A$ < > "U" AND A$ < > "M"
AND A$ < > "K" AND A$ < > "H"
AND A$ < > "J" THEN  GOTO 130
150  IF A$ = "E" THEN  GOSUB 1010
160  IF A$ = "E" THEN  GOTO 130
170  IF A$ = "F" THEN  GOTO 320
180  IF A$ = "U" THEN D = 0:Y = Y - 5
190  IF A$ = "M" THEN D = 2:Y = Y + 5
200  IF A$ = "K" THEN D = 1:X = X + 5
210  IF A$ = "H" THEN D = 3:X = X - 5
220  IF A$ < > "J" THEN  GOTO 250
230  IF O = 1 THEN O = 0: GOTO 130
240 O = 1: GOTO 130
250  IF O = 1 THEN  HPLOT  TO X,Y:
GOTO 270
260  HPLOT X,Y
270 A%(P,1) = D:A%(P,2) = O
280 P = P + 1: IF P = 101 THEN
GOTO 320
290  GOTO 130
300  REM
310  REM  SET UP FOR MAKING THE TABLE
320  TEXT
330  PRINT : PRINT
340  PRINT "NOW MAKING UP THE TABLE"
350  DIM B%(101)
360 PA = 1:PB = 1: REM  POINTERS TO THE
ARRAYS
370 P = P - 1: REM   NOW MARKS LAST
ENTRY IN A%(,)
380 PR = 1: REM  MARKS WHICH PAIR IN THE
BYTE WE ARE AT
390  REM
400  REM  MAKE UP THE TABLE
410  REM  NEXT ONE
420  IF PA > P THEN  GOTO 530
```

```
430 O = A%(PA,2)
440 D = A%(PA,1)
450 D = D + 4 * O
460  IF PR = 1 THEN B%(PB) = D:PR = 2:PA
= PA + 1: GOTO 410
470  IF PR = 2 THEN B%(PB) = B%(PB) +
  8 * D:PA = PA + 1:PR = 3: GOTO 410
480  IF D > 0 AND D < 3 THEN B%(PB)
= B%(PB) + 64 * D:PA = PA + 1:PB = PB +
  1:PR = 1: GOTO 410
490  IF B%(PB) > 7 THEN PB = PB + 1:
PR = 1: GOTO 410
500 B%(PB) = B%(PB) + 64:PB = PB + 1:
PR = 2 :B%(PB) = 3: GOTO 410
510  REM
520  REM  END OF THIS PART
530  IF PR = 1 THEN PB = PB - 1:
GOTO 600
540  IF PR = 2 THEN  GOTO 580
550  IF B%(PB) > 7 THEN  GOTO 600
560 B%(PB) = B%(PB) + 64
570 PB = PB + 1:B%(PB) = 3: GOTO 600
580  IF B%(PB) > 0 THEN  GOTO 600
590 B%(PB) = B%(PB) + 200
600 PB = PB + 1
610 B%(PB) = 0
620  REM
630  REM  PUT THE RESULTS ONTO THE
SCREEN
640  FOR I = 1 TO PB
650 A = B%(I)
660  GOSUB 1220
670  PRINT A$
680  NEXT I
690  REM
700  REM  SAVE RESULTS
710  PRINT
720  INPUT "CHOOSE WHICH TYPE OF OUTPUT
(S=SCREEN, T=TAPE, D=DISC) ";A$
730  IF A$ = "S" THEN  GOTO 790
740  IF A$ = "T" THEN  GOTO 970
```

```
750  IF A$ = "D" THEN  GOTO 850
760  GOTO 710
770  REM
780  REM  OUTPUT TO SCREEN
790  FOR I = 1 TO PB
800  PRINT B%(I),
810  NEXT I
820  STOP
830  REM
840  REM  OUTPUT TO DISC
850  PRINT : PRINT  CHR$ (4);"CATALOG"
860  PRINT : INPUT "WHAT OUTPUT FILE
NAME ? ";O$
870  PRINT  CHR$ (4);"OPEN ";O$
880  PRINT  CHR$ (4);"WRITE ";O$
890  FOR I = 1 TO (PB - 1)
900  PRINT B%(I)
910  NEXT I
920  PRINT B%(PB)
930  PRINT  CHR$ (4);"CLOSE ";O$
940  STOP
950  REM
960  REM  OUTPUT TO TAPE
970  INPUT "SET TAPE TO RECORD, THEN
PRESS RETURN ";A$
980  STORE B%
990  STOP
1000  REM
1010  REM  ROUTINE TO ERASE AN ENTRY AND
RE-DRAW
1020  HGR
1030  HCOLOR= 3
1040 P = P - 1
1050 X = 140:Y = 80
1060  HPLOT X,Y
1070  FOR I = 1 TO P - 1
1080  ON A%(I,1) GOTO 1110,1130,1150
1090 Y = Y - 5
1100  GOTO 1160
1110 X = X + 5
1120  GOTO 1160
```

```
1130 Y = Y + 5
1140  GOTO 1160
1150 X = X - 5
1160 IF A%(I,2) = 1 THEN HPLOT TO X,Y
1170 IF A%(I,2) = 0 THEN HPLOT X,Y
1180  NEXT I
1190  RETURN
1200  REM
1210  REM  ROUTINE TO CONVERT DECIMAL TO
BINARY
1220 A$ = ""
1230  FOR J = 1 TO 8
1240 B = A -  INT (A / 2) * 2
1250  IF B > .5 THEN A$ = "1" + A$
1260  IF B < .5 THEN A$ = "0" + A$
1270 A =  INT (A / 2)
1280  NEXT J
1290  RETURN
```

Line 10 dimensions the array A%(,) which will hold the incoming shape. There is room in A%(,) for 100 steps, and each step has two parts. The first part A%(,1) holds the *direction* of the segment, using the same code that the Apple uses:

Direction	*Value*
Up	0
Right	1
Down	2
Left	3

If you look at the table given earlier in this chapter, you will find that these are the decimal equivalents of the binary 'direction' codes used in the shape table. The

second part, A%(,2), indicates whether this is a *move* or a *draw*. If it's just moving, then it will hold 0, but if it's drawing it will hold 1.

Lines 20 to 30 set up the screen (we don't need to worry about ROT and SCALE, as we're only going to use HPLOT, not DRAW or XDRAW).

Lines 40 and 50 set up the starting point for the 'shape' on the screen. It's line 40 that gives the dot in the middle of the screen when you start.

Line 60 sets the initial value of P (the 'pointer' to A%(,)). The value of P will always be the next empty entry in A%(,). At the start of the program, A%(,) is empty, and so A%(1,) is the first available entry.

Line 70 sets the O variable — this keeps track of whether the current state of the line is 'on' (O=1) or 'off' (O=0). Again, this is the same code as used with the shape table in the Apple.

Lines 80 to 100 print the message on the text part of the screen that reminds you what each of the keys is for.

Line 130 takes in one key press — it doesn't show the letter on the screen, and it doesn't need the RETURN key to be pressed. So every time you press a key, the A$ variable will be set to it.

Line 140 simply tests that the key you pressed was one of the ones that is allowed. If it is not (for example, if you'd pressed the X key by mistake), then line 140 would send the program back to line 130, ignoring the erroneous key.

Line 150 notices when the E key has been pressed (which asks the computer to erase the last entry). It sends the program to the erasing subroutine at line 1010. After the routine has been done, the computer will go to line

160. A$ will still be "E", and so line 160 will send it back round the loop again for the next key.

Lines 180 to 210 notice the four direction keys. For each one, the value of either X or Y is altered. In the same way as in the LANDER program, X and Y hold the horizontal and vertical positions on the screen. The value of D (which holds the 'direction' code) is also set in these four lines. If the key that was pressed was J, then line 220 will send the program to lines 230 and 240. J, you remember, turned the line 'on' or 'off' — the current state being stored in variable O.

Lines 230 and 240, although looking rather peculiar, set the value of O to 0 if it was 1, and to 1 if it was 0.

If the key that had been pressed was one of the direction ones, then line 220 will send the program to line 250.

Line 250 will draw a solid line to the new position (represented by the new values of X and Y). The operation of the HPLOT command is dealt with in depth in the next chapter.

If O=0 (i.e. if the line is 'off'), then line 260 will be done, and this will put a dot at the new position, leaving no line from the old position.

Line 270 puts the step into A%(,), so that the steps can be analysed later in the program to produce the table.

Line 280 increases the value of P (so that it now 'points' to the next entry — which will be the next available one). Line 280 also checks to see whether A%(,) is full (which will happen when P=101). This will only occur if you try to put in a shape which has over 100 steps.

Line 290 sends the program back round the loop again.

All of this will go on until you press the F key to tell the computer that the shape is finished.

Erasing

The erasing subroutine starts at line 1010.

At line 1020, an HGR command clears the top part of the screen.

Line 1030 makes sure that the HGR command has not had an effect on the current color.

Line 1040 decreases the value of the 'pointer', effectively taking one of the entries off the 'top' of A%(,). The last step that you had put in is now the 'next available' space in A%(,). This means that it has effectively been erased.

Line 1050 sets X and Y back to their original positions, ready to re-draw the shape up to (but not including) the erased step.

Line 1060 puts the dot in the middle of the screen again.

Lines 1070 to 1180 go through all of the entries in A%(,), plotting them onto the screen one by one, and updating the values of X, Y and O as it goes. What this is doing is almost like re-entering each of the steps from the keyboard one by one.

Line 1190 is the end of the 'erase' routine. The computer will then go back to line 160.

By the time we reach line 310, we have all of the necessary values in A%(,), and we're ready to produce the table entries.

Line 320 sets the screen back to 'text' mode, and the high-resolution graphics will disappear.

Line 350 defines the array B%(), which will hold the actual shape table entries. What we have to do now is to convert the 'moves' in array A%(,) into the entries in array B%().

Line 360 defines the 'pointers' to the two arrays — PA points to A%(,), while PB (no surprise) points to B%(). What this means is that these two variables (PA and PB) will keep track of where we are in each array, as we go through the moves in A%(,) and translate them into values for B%().

Line 370 sets P, which had pointed to the 'next available entry' in A%(,), to the 'last entry' in A%(,). So the entries in A%(,) run from 1 to P.

Line 380 sets PR, which 'points' to the particular part of the B%() entry that we're working on. Remember that each shape table entry is really *three* 3-digit binary numbers? Well, PR will hold 1, 2 or 3, depending on which of those 3-digit numbers we're on at present.

Line 410 marks the start of the loop which goes through each entry in A%(,), translating it into its place in B%().

Line 420 checks to see whether we've reached the last entry in A%(,). P is the number of the last entry, and PA is the number of the entry we're currently working on.

Line 430 sets O to the 'draw or move' part of A%(PA,). O will be 1 if this is a 'draw', or 0 if this is a 'move'.

Similarly, line 440 sets D to the direction code (0 to 3). In fact, variables O and D are being used here in exactly the same way as when recording the moves earlier in the program.

Line 450 makes D equal to the binary equivalent of the three-digit group. D represents a two-digit binary number between 0 and 3 (which are 00 and 11 in binary). What we want to do is to put O in the 'most significant' position in that group. We want to add O onto the 'top' of that two-digit group. In the three-digit codes used in the shape table, the first bit is the one represented by O — 0 is move, 1 is draw. The next two bits are the 'direction' bits. By multiplying O by 4, and then adding it to the value in D, we're really putting O in as the third digit. For example:

```
Value of D=2 . . . which is 10 in binary.
Value of O=1 . . . which is 1 in binary.
Value of D+4*O is 2+(4X1)=6.
New value of D is 6 . . . which is 110 in binary.
```

So you can see that by adding 4*O to the value of D, we've added a 1 to the top of the previous value of D — 10, giving 110. If you don't fully appreciate this point, it's probably worth working through some more examples yourself.

Line 460 takes care of the case where we're dealing with the first 3-bit group (that's the right-hand one, as the groups are looked at from right to left). What happens is very simple. We set B%() to D, which is the three-bit group we've just worked out. Then we set PR so that it tells us we're working on the next group in that entry

(PR=2), then we increase PA by 1, taking us to the next entry in A%(,), then we jump to the start of the loop at line 410. This is all we have to do for the case where PR=1.

Line 470 does the same thing for the case where PR=2. The only difference is that instead of just setting B%() to D, we multiply D by 8, and then *add* it to the current value of B%(). This has a similar effect to what we did with O and D earlier. It sets the fourth, fifth and sixth bits of B%() to the current value of D. Again, if you can't follow it, try a couple of examples for yourself.

By the time we get to line 480, we know that PR is 3 (it can't be 1 or 2, and it won't be 4 or more).

Bearing in mind that the computer will ignore any groups that are 000, and which have only zeroes to the left of them, and will treat any entry which is 00000000 as the end of the shape, we have to watch for the case where there are a number of genuine 000 (i.e. 'move up') entries in a row. Line 480 deals with cases where D is not 000 (in other words, it will not cause trouble with a preceding 000 group) and where the top bit of D is off (which, remember, is the only case allowed in the third group in an entry — there being only two digits available). D is simply added to the top of the current B%() entry, and both PA and PB are increased (putting us in the next entry of A%(,), and a fresh entry in B%(). PR is, of course, set back to 1 for the first group in the new shape table entry.

Line 490 deals with cases where D is 000 or has its top bit on. If the value already in B%() is greater than 7, then we are sure that the second group has at least one bit on. We are then quite safe in leaving that entry in B%() as it is and going on to the next one. The top group will then

be 00 — but it will be ignored by the computer as it is all zeros.

Line 490 deals with the really nasty case where D is 000 or has its top bit on, and at the same time B%() has 000 as its second group. We get around this problem by inserting two 'dummy' moves of 'move right' (001) and 'move left' (011). The 001 goes at the top of the current B%() value, and the 011 goes at the beginning of the next one. When the computer goes to draw the shape, these two moves (although the computer will actually do them) will not affect the shape. At the same time, however, the second group of the current B%() will be preserved, as there is a bit on to the left of it.

When all of the entries of A%(,) have been dealt with, line 420 will send the program to lines 530 to 590, which put in the second-last entry in B%().

Lines 600 and 610 put in the last entry in B%(), which, of course, is 0.

Lines 640 to 680 use the subroutine at line 1220 to put the results onto the screen in binary form. The subroutine at line 1220 (which is one of the tasks set in Appendix 1), works on the principle that odd numbers always have a least-significant binary digit of 1. The number (held in variable A) is divided again and again at line 1270 (the INT function just removes the fractional part, leaving A as a whole number).

Line 1240 sets B equal to 1 if A is odd, and 0 if it is even.

Lines 1250 and 1260 make up a string, A$, which is the binary result. The string concatenation function, '+', simply adds the new digit to the left of the current value of A$. "AA"+"BB" will give "AABB", and so on.

The rest of the program (from 700 to 990) is fairly

straightforward, putting the results out to the screen, to tape or to disc (depending on what you have available). Note that it is not necessary to record how many entries there are, as the last one is always 0.

The MAKE SHAPE program allows you to make up one shape. The COMBINE SHAPES program automatically combines a number of shapes into one shape table, stores the shape table in memory, and generates a program segment which can later set up the shape table from a different program.

Using COMBINE SHAPES

When the COMBINE SHAPES program is run, it will generate a series of messages similar to the following:

```
THE NUMBER OF SHAPES STORED IN THE MACHINE
IS 1
THE TABLE IS IN MEMORY FROM 38391 TO
38398
HIMEM: IS SET AT 38390
```

These refer to the current status of the shape table in memory. The numbers are the addresses that the shape table falls in (from 38391 to 38398) and the current value of HIMEM: (38390). HIMEM: is set *below* the start of the table to protect it from being over-written by BASIC.

Further messages ask you to either select where the next shape is to come from, or to do a couple of other things.

```
WHAT IS THE SOURCE OF THE NEW SHAPE:
K=KEYBOARD, T=TAPE, D=DISC, F=FINISH,
E=ERASE, O=OUTPUT?
```

Entering F will cause the program to stop, leaving the shape table intact, so that you can use it.

Entering E will erase the current shape table, and cause the message:

```
NO SHAPES IN THE MACHINE AT PRESENT
HIMEM: IS AT 38400
```

The shape table has been set back to the way it would have been when the computer was turned on.

Entering K, T or D will cause the program to take the next shape from either the keyboard (entries to be typed in one by one), from tape, or from disc. The shapes will, of course, be in the format produced by the MAKE SHAPE program. As each shape is read in (any number can be used), the program will put it in as shape number 1, moving all the *rest* of the shapes up by 1. So if you have three shapes and want them to be shapes number 1, 2 and 3, then put them in in the order 3, then 2, then 1.

Using the F command at this stage will cause the program to finish, so that you can use the HGR and DRAW commands to put the shape onto the screen. Don't forget to set the ROT and SCALE variables. You can then RUN the program again. It will not erase the shape table until you use the E command, so you can add more shapes at this stage if you wish. When you are happy with the shape table, you can produce a permanent record of it using the 'O' command. This generates and outputs a

segment of program, starting at line number 5000, which (when run) will put the current shape table into memory.

The 'output' command will also place the cursor (the flashing square) at the top of the screen. One of the editing features of the Apple is that, when you move the cursor over anything on the screen, it is as if you had typed it. So when the lines of program appear on the screen, you can use the forward arrow key (→) to move the cursor over each of the lines in turn to the end of the line, pressing the RETURN button to enter each line into memory. What you will end up with is a program that starts at line number 5000, which, when run, will put the shape table into memory.

Meanwhile, the COMBINE SHAPES program will have erased itself (!), leaving just the new program in memory. Having got the program segment into memory, it's simply a matter of saving it (on tape or disc) for future use.

When you come to write the rest of the program which will use the shape table, you start by putting the program segment into memory. The rest of the program will, of course, have to use line numbers under 5000. All you have to do then is to arrange for the program to first call lines 5000 onwards as a routine. For example see the LANDER program earlier in this chapter. The routine at the end of the LANDER program was produced using the COMBINE SHAPES program.

COMBINE SHAPES

```
10  DIM B%(101)
20 HI =  PEEK (116) * 256 +  PEEK (115):
REM  HI IS NOW CURRENT HIMEM: SETTING
30 ST =  PEEK (233) * 256 +  PEEK (232):
REM  ST IS NOW STARTING ADDRESS OF
SHAPES TABLES
40  PRINT
50 FL = 0
60  IF ST = 65535 THEN  PRINT "NO SHAPES
IN THE MACHINE AT PRESENT": PRINT :
PRINT "HIMEM IS ";HI:FL = 1: GOTO 220
70 NS =  PEEK (ST)
80  PRINT
90  PRINT "THE NUMBER OF SHAPES STORED
IN THE MACHINE IS ";NS
100 P = ST: REM  POINTER
110 P = 2 + P: REM  GETS IT PAST THE
NUMBER OF SHAPES
120 P = P + 2 * (NS): REM  NOW AT THE
START OF THE FIRST SHAPE
130 N = NS
140  FOR P = P TO 65535
150  IF  PEEK (P) = 0 THEN N = N - 1:
REM  END OF ANOTHER SHAPE
160  IF N < .5 THEN  GOTO 180
170  NEXT P
180 EN = P: REM  EN IS THE TOP OF THE
TABLE
190  PRINT : PRINT "THE TABLE IS IN
MEMORY FROM ";ST;" TO ";EN
200  PRINT "HIMEM: IS SET AT ";HI
210 LN = EN - ST + 1: REM  LENGTH OF
EXISTING TABLE
220  PRINT : PRINT "WHAT IS THE SOURCE
OF THE NEW SHAPE:"
230  INPUT "K=KEYBOARD, T=TAPE, D=DISC,
F=FINISH, E=ERASE, O=OUTPUT ";A$
240  IF A$ = "O" THEN  GOTO 1060
250  IF A$ = "K" THEN  GOTO 310
```

```
260  IF A$ = "T" THEN  GOTO 410
270  IF A$ = "D" THEN  GOTO 500
280  IF A$ = "F" THEN  STOP
290  IF A$ = "E" THEN  HIMEM: 38400:
POKE 233,255: POKE 232,255: GOTO 20
300  GOTO 220
310  REM  ***** ENTER FROM KEYBOARD
320  PRINT : PRINT "ENTER THE VALUES ONE
BY ONE, FINISHING (OF COURSE) WITH A 0"
330  PRINT
340  FOR I = 1 TO 101
350  PRINT I;" ";: INPUT B%(I)
360  IF B%(I) = 0 THEN  GOTO 390
370  NEXT I
380  PRINT "SORRY - ONLY 101 ENTRIES
ALLOWED (INCLUDING THE 0)": STOP
390 PB = I
400  GOTO 630
410  REM  ****** ENTER FROM TAPE
420  PRINT : INPUT "GET TAPE READY, THEN
PRESS RETURN ON COMPUTER, THE PLAY ON
TAPE RECORDER ";A$
430  RECALL B%
440  FOR I = 1 TO 101
450  IF B%(I) = 0 THEN  GOTO 480
460  NEXT I
470  PRINT "ERROR - INPUT DOES NOT HAVE
A 0 AT THE END OF IT": STOP
480 PB = I
490  GOTO 630
500  REM  ***** ENTER FROM DISC
510  PRINT : PRINT  CHR$ (4);"CATALOG"
520  PRINT : INPUT "WHAT FILE NAME ? ";
I$
530  PRINT  CHR$ (4);"OPEN ";I$
540  PRINT  CHR$ (4);"READ ";I$
550  FOR I = 1 TO 101
560  INPUT B%(I)
570  IF B%(I) = 0 THEN  GOTO 600
580  NEXT I
```

```
590  PRINT  CHR$ (4);"CLOSE ";I$:
PRINT "INPUT DOES NOT CONTAIN A 0, OR IS
TOO LONG ": STOP
600 PB = I
610  PRINT  CHR$ (4);"CLOSE ";I$
620  GOTO 630
630  REM  ******* PUT SHAPE INTO MEMORY
640  REM  THE SHAPE IS IN B%(), AND PB
IS THE NUMBER OF ELEMENTS
650  IF FL = 1 THEN  GOTO 920: REM  NO
SHAPES IN THE MACHINE - ADD A FRESH ONE
660  REM  FIRST, MOVE THE EXISTING
SHAPES DOWN TO MAKE ROOM
670 HI = HI - 2 - PB: REM  MAKE ROOM FOR
NEW INDEX, TOO
680  HIMEM: HI
690  FOR I = HI + 1 TO EN - PB
700  POKE I, PEEK (I + PB)
710  NEXT I
720 ST = ST - 2 - PB
730 POKE 233, INT (ST / 256)
740 POKE 232,ST - 256 * INT (ST / 256)
750 NS = NS + 1
760  POKE ST,NS
770  POKE ST + 1,0
780  POKE ST + 2,LN + 2
790  POKE ST + 3,0
800 P = ST + 2 + LN
810  REM  NOW PUT IN THE NEW SHAPE,
STARTING AT ADDRESS P
820  FOR I = 1 TO PB
830  POKE P + I - 1,B%(I)
840  NEXT I
850  REM  NOW UPDATE THE VALUES IN THE
OTHER INDEXES
860  IF NS = 1 THEN  GOTO 910: REM  FOR
THE FIRST SHAPE
870  FOR I = 1 TO (NS - 1)
880 P = ST + 4 + 2 * (I - 1)
890  POKE P, PEEK (P) + 2
```

```
900  NEXT I
910  GOTO 20
920  REM  ******* PUT IN A SHAPE WHERE
NO OTHER SHAPE HAS EVER GONE BEFORE ...
930 HI = HI - PB - 6
940 ST = HI + 1
950  POKE ST,1
960  POKE ST + 1,0
970  POKE ST + 2,4
980  POKE ST + 3,0
990  FOR I = 1 TO PB
1000  POKE ST + I + 3,B%(I)
1010  NEXT I
1020  HIMEM: HI
1030 POKE 233, INT (ST / 256)
1040 POKE 232,ST - 256 * INT (ST / 256)
1050  GOTO 20
1060  REM  ****** PUT A PROGRAM SEGMENT
ONTO THE SCREEN IN SUCH A WAY THAT IT
CAN BE ENTERED BY USING THE EDITING
COMMANDS
1070  HOME
1080  PRINT
1090  PRINT " 5000 HIMEM: ";HI
1100  PRINT
1110 PRINT " 5010 POKE 232,";PEEK (232); " :
POKE 233,"; PEEK (233)
1120  PRINT
1130  PRINT|" 5020 FOR I=1 TO ";LN
1140  PRINT
1150  PRINT|" 5030 READ N"
1160  PRINT
1170  PRINT " 5040 POKE ";HI;"+I, N"
1180  PRINT
1190  PRINT " 5050 NEXT I"
1200  PRINT
1210  PRINT " 5060 DATA ";
1220  FOR I = 1 TO LN
1230 T = 0
1240 IF I = 10 * INT (I / 10) THEN T = 1
1250  PRINT  PEEK (HI + I);
```

```
1260  IF T = 1 THEN  PRINT : PRINT :
PRINT " ";5060 + I;" DATA ";
1270  IF T = 0 THEN  PRINT ",";
1280  NEXT I
1290  VTAB 1
1300  REM  NOW DELETE THE REST OF THE
PROGRAM!
1310  DEL 0,4999
```

Line 10 defines the array that's going to hold the incoming shapes. Naturally, if 101 entries isn't enough for you, you can increase it simply by altering both programs.

Line 20 sets the variable HI to the current value of HIMEM:.

Line 30 does the same thing for the starting address of the shape table.

Line 50 sets the flag variable FL to 0.

Line 60 checks to see whether the 'start' of the shape table is at address 65535. This is actually the highest possible address in the Apple. If the shape table starts there, then it means in effect that there is no shape table! The Apple, when you turn it on, will set the start of the shape table to 65535, indicating that there is no table in memory.

Line 60 prints a message to say that the table is empty, and sets FL to 1, indicating to the rest of the program that this is the case.

Line 70 sets NS to the contents of the first entry of the shape table, which is the *number* of shapes in the table (NS = number of shapes).

Line 100 sets a 'pointer' variable, P, to the start of the table. Line 110 moves it past the first two bytes (which are the number of shapes and a zero, respectively).

Line 120 puts the pointer past the parts of the table that tell the computer where each of the shapes start. Each of these is two bytes long, and there will be twice as many of these entries as there are shapes.

By the time we get to line 130, P will be at the start of the first shape. Line 130 sets a variable N to the number of shapes. The point of all this is to find the end of the shape table, and to check the number of 0s in it.

Line 140 sets P going up in steps from its current value (at the start of the first shape) to the end of memory.

Line 150 counts the number of 0s in the table by reducing N every time it meets one.

Line 160 will stop the loop when N is zero. When this is the case, we must have reached the end of the table, or the end of memory.

Line 180 sets variable EN to the address of the end of the table.

We now have ST set to the start of the table, HI set to the value of HIMEM: and EN set to the address of the end of the table.

Line 210 sets LN to the *length* of the table.

Lines 220 to 300 accept the next command, and send the computer to the appropriate part of the program.

Line 290 will empty the shape table by setting each of the addresses that are used to store the start of the table to 255. This will give a resulting address for the start of 256*255+255, which is 65535.

The parts of the program that read the shapes are pretty straightforward (lines 310 to 620). Basically they are the reverse of the corresponding parts of the MAKE SHAPE program.

When we get to line 640, B%() holds the shape to be added to the table, and PB holds the number of entries in B%() (including the 0 at the end).

Line 650 checks the flag, FL, that was set to 1 in line 60 if there was no table in memory. If FL is 1, then line 650 will send the computer to line 920.

Lines 920 to 1050 set up a shape in the table when none is there already. That's simply a matter of setting the various variables to the right values and POKEing the shape into memory. The only clever part of this section is lines 1030 and 1040, which work out the two values needed for the start address of the table. Remember, the address is stored as two numbers, the first being the number of 256s.

If the shape is *not* the first one in the machine, then lines 660 to 910 are used.

Line 670 works out the new value of HIMEM:. Lines 690 to 710 move the existing shapes *down* in memory to make room for the new one.

Line 720 works out the new starting address, and lines 730 and 740 store it in memory.

Lines 750 to 790 set up the start of the table, putting the new shape in as shape number 1, and setting up the new value of NS, the number of shapes.

Lines 800 to 840 actually POKE the new shape into memory (including the 0 at the end).

Lines 850 to 900 alter the 'start of shape' entries for the other shapes.

Finally, the segment from line 1070 to 1310 is the 'output' function of the program. It generates a program segment that will put the complete shape table into memory when it is run later.

Line 1070 clears the screen and places the cursor at the top left corner.

Lines 1090 to 1210 output the lines of program on the screen, separated by a blank line, when you press RETURN at the end of each line.

Lines 1220 to 1280 output the shape table in the form of DATA statements. All the fancy footwork here is just to ensure that you get 10 entries per statement, and that the last one doesn't have a comma after it.

Lastly, line 1290 returns the cursor to the top left corner of the screen, and line 1310 deletes the rest of the program!

Figures in Two Dimensions

Chapter 2

Although the shape-handling commands of the Apple are very fast — fast enough for games, etc. — they will not serve us very well for the next stage of our journey into graphics.

The HPLOT command which allows us to draw lines on the screen from point to point can be used to draw a 'figure' (as opposed to a 'shape') which is made up of individual lines. This chapter will take you through all of that, finishing with a suite of programs which will allow you to generate animations automatically.

The sort of animation that we'll be using is not fast enough for games and the like but can be used to produce true animations using film and other media.

HPLOT takes one of two forms, in the same way that the DRAW command does. The first specifies the starting *and* finishing positions, and the second uses the previous finishing position as the new starting position. A position on the screen is specified in terms of a pair of numbers in the same way as the DRAW command: the first number is

the distance from the left of the screen; and the second number the distance from the top of the screen. Let's look at an example:

```
HPLOT 5,10 TO 100,120
```

This command specifies two screen positions — 5,10 and 100,120 (the first is near the top left of the screen, and the second is near the middle) and will draw a line from the first to the second. The color of the line will be the last color set by the HCOLOR command. Alternatively a command like HPLOT TO 150,80 will plot a line from the last HPLOTted position to the one given here, i.e. 150,80.

A variation of this form of the HPLOT command uses a whole series of points drawing a line to each point in turn from the last one mentioned — as in the following example which will draw a square.

```
HPLOT 10,10 TO 90,10 TO 90,90 TO 10,90 TO
10,10
```

Graphs

We can use HPLOT to draw graphs and other charts. Take a look at the following program, for example:

GRAPH

```
10  HGR
20  HCOLOR= 3
30  HPLOT 10,10 TO 10,140 TO 140,140
40  FOR I = 1 TO 10
50  PRINT "VALUE (0 TO 10) ";
60  INPUT A
70 IF I = 1 THEN HPLOT 10 + (I - 1) * 10,
   140 - 10 * A
80  IF I < > 1 THEN  HPLOT  TO 10 + (I
- 1) * 10,140 - 10 * A
90  NEXT I
```

This program accepts as input a series of 10 numbers (each between 0 and 10) and draws a graph accordingly. For example, using the numbers 3,1,4,1,5,9,2,6,5,3, it will generate the graph shown in Figure 2–1.

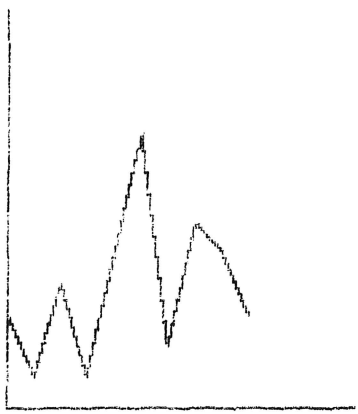


Figure 2–1

Lines 10 and 20 in this program should be familiar to you by now — they set up the screen so that the top part shows high-resolution graphics and the bottom few lines show text. (To get the screen to revert to its normal all-text mode, simply type in the command TEXT.)

Line 30 draws the axes (the two lines that make up the border on the bottom and left of the graph). The axes will start at the point 10,10 (near the top left of the screen), will go down vertically to 10,140 (at the left near the bottom) and then across to 140,140 (near the bottom of the screen in the middle).

Lines 40 and 90 are simply a loop for entering the 10 numbers.

Lines 50 and 60 assign the value of each number as it is read to the variable A.

The start of line 70 will detect when I=1, in other words, if this is the *first* point of the graph. In that case, we have no previous point to draw *from*, so we use HPLOT rather than HPLOT TO. For all subsequent points, though, we want to draw a line from the previous point to the new point.

The equation towards the right of line 80 is a way of finding out the position of the new point. The first part (before the comma) will determine how far from the left of the screen the new point will lie. That is going to change according to I which starts at 1 and steps through to 10 as each point is entered. The first co-ordinate of the new position (the horizontal distance) is calculated by this equation: 10+(I−1)*10. It subtracts 1 from I, multiplies by 10, then adds 10. This will give (as the value of I goes from 1 to 10) a position that starts at 10 (when I is 1) and goes to 110 (when I is 10). So, as I goes from 1 to 10,

the position of each point across the screen will step (in steps of 10) from 10 to 110.

The second co-ordinate of the new position (which gives the distance down from the *top* of the screen) is calculated by the equation: 140−10*A, which multiplies A (which is the number we've just entered) by 10, and then subtracts that answer from 140. As A varies from 0 to 10, this will give a distance from the top of the screen that varies from 140 (the bottom of the graph) to 40 (which is near the top of the screen).

Line 80 then calculates the co-ordinates of the new point and causes a HPLOT TO that point from the last point plotted.

To make the graph look more interesting, we can add little 'markers' at each point plotted, using the shape drawing facilities as discussed in the last chapter (see Figure 2–2).

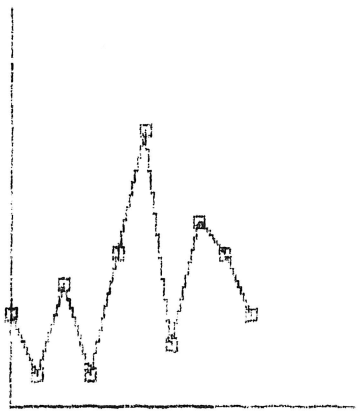


Figure 2–2

The program that produced this graph is very similar to the GRAPH program. It is listed below.

GRAPH+POINTS

```
10  HGR
20  GOSUB 5000
30  SCALE= 2
40  HCOLOR= 3
50  HPLOT 10,10 TO 10,140 TO 140,140
60  FOR I = 1 TO 10
70  PRINT "VALUE (0 TO 10) ";
80  INPUT A
90  IF I = 1 THEN  HPLOT 10 + (I - 1) *
    10,140 - 10 * A
100  IF I < > 1 THEN  HPLOT  TO 10 +
(I - 1) * 10,140 - 10 * A
110  DRAW 1
120  NEXT I
130  STOP
5000  HIMEM: 38388
5010  POKE 232,245: POKE 233,149
5020  FOR I = 1 TO 10
5030  READ N
5040  POKE 38388 + I,N
5050  NEXT I
5060  DATA  1,0,4,0,35,45,54,63,12,0
5070  RETURN
```

Line 20 sends the program to the subroutine at line 5000 (produced automatically using the COMBINE SHAPES program listed in the last chapter) which sets shape number 1 to a small square.

Line 30 sets the scale to 2, line 40 sets the color to white.

The rest of the program is the same as GRAPH, except of course that the line numbers are different, and that line 110 has been added.

Line 110 draws shape number 1 at each of the points on the graph. As discussed earlier, it is usual in this sort of

work to arrange (when using the MAKE SHAPE program) the shapes to start and finish at the same point, so that the DRAW command does not affect the position of each successive line as the graph is HPLOTted.

Histograms

Histograms are becoming a popular method of presenting a set of values or results. One reason for this is that our eyes read the shape of the blocks in a histogram more easily than dots or lines. Figure 2–3 shows a histogram produced with the same series of numbers as was used for the graphs, by the program 'HISTOGRAM'.

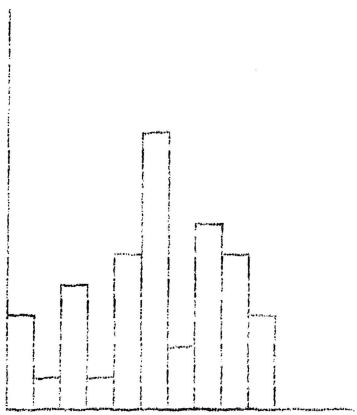


Figure 2–3

HISTOGRAM

```
10  HGR
20  HCOLOR= 3
30  HPLOT 10,10 TO 10,140 TO 140,140
40  FOR I = 1 TO 10
50  PRINT "VALUE (0 TO 10) ";
60  INPUT A
70  HPLOT 10 + (I - 1) * 10,140
80  HPLOT TO 10 + (I - 1) * 10,140 -
    10 * A
90  HPLOT TO 10 + I * 10,140 - 10 * A
100  HPLOT TO 10 + I * 10,140
110  NEXT I
```

This program is very similar to the GRAPH program, except that lines 70, 80, 90 and 100 replace the plotting parts of the GRAPH program.

Line 70 starts the HPLOT at the bottom left of the 'bar' of the histogram.

Line 80 draws the left-hand side of the bar, line 90 the top and line 100 the right-hand side.

Where successive bars overlap, some parts of the screen will be HPLOTted twice.

To make the histogram look more impressive, we can give it a 'solid' look by drawing blocks rather than rectangles (see Figure 2–4).

SOLID HISTOGRAM

```
10  HGR
20  HCOLOR= 3
30  HPLOT 10,10 TO 10,140 TO 140,140
40  FOR I = 1 TO 10
50  PRINT "VALUE (0 TO 10) ";
```

```
60  INPUT A
70 X = 10 + (I - 1) * 10
80 Y = 140 - 10 * A
90  FOR J = 0 TO 10
100  HPLOT X + J,140 TO X + J,Y
110  NEXT J
120  REM  NOW FOR THE BACK
130  HPLOT X,Y TO X + 5,Y - 5
140  HPLOT  TO X + 15,Y - 5
150  HPLOT  TO X + 15,140
160  HPLOT X + 10,Y TO X + 15,Y - 5
170  NEXT I
```

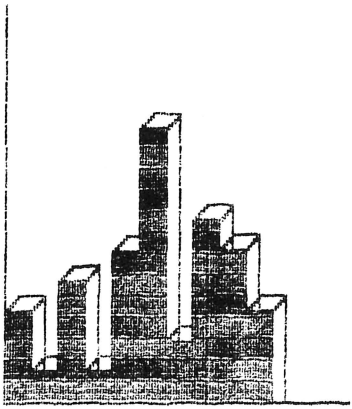


Figure 2–4

Lines 70 and 80 calculate the position of the top left corner of the 'front' of each bar using the same methods as were used for calculating the graph points earlier. These are calculated here to simplify things later on, where we'll be using the values of X (the distance from the top of the screen) and Y (the distance from the left of the screen) quite a lot.

Lines 90, 100 and 110 are a loop which draws lines from the bottom of each of the bars to the top. The lines are drawn as close together as possible giving the impression of solid color to the front of the bar.

Lines 130 to 160 draw in the back section of each bar. Line 130 draws the left top edge, line 140 the top back edge, line 150 the right back edge and line 160 the top right edge.

Plotting curves

The drawing of graphs is not limited to a series of individual points. Using the mathematical functions of BASIC, we can draw graphs of mathematical equations.

Figure 2–5 is a graph of the function $e^{x/5}\sin(3x)$ (for the mathematically minded amongst you!) as generated by the CURVE PLOT program.

CURVE PLOT

```
10  HGR
20  HCOLOR= 3
30  HPLOT 10,70 TO 250,70
40  HPLOT 20,10 TO 20,150
50 X = 0
60  GOSUB 1000
70  HPLOT 20,Y
80  FOR X = 0 TO 10 STEP 0.05
90  GOSUB 1000
```

```
100  HPLOT  TO X * 22 + 20,Y * 7 + 70
110  NEXT X
120  STOP
1000 Y =  EXP (X / 5) *  SIN (X * 3)
1010  RETURN
```

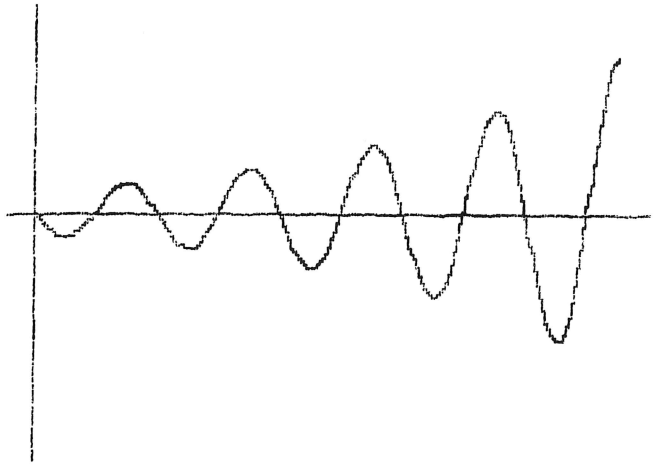


Figure 2–5

Lines 10, 20 and 30 set up the screen and draw the axes.

The subroutine starting at line 1000 is the 'function' that we are trying to plot. Change the equation at line 1000 to plot the graph of other mathematical equations. Whenever the subroutine is called, it will take the value of X (the distance from the left of the screen) and find the corresponding value of Y (the distance from the top of the screen), with X=0 and Y=0 being at the origin of the axes (i.e. the place where the two straight lines cross).

Line 50 sets X to the start of the graph, line 60 finds the corresponding value of Y, and line 70 HPLOTs the first point of the graph. As in the previous graph example, we have to HPLOT the *first* point separately, otherwise we get a line from the last thing that we drew.

Having HPLOTted the first point, we can use 'HPLOT TO' to draw a line from the previously plotted point to the new point.

Lines 80 and 110 increase X from 0 to 10 in steps of 0.05. Line 90 calculates the corresponding value of Y, and line 100 calculates and HPLOTs TO the correct position for the point on the screen.

3-D in two dimensions

An example of sophisticated graph plotting is a graph of a three-dimensional surface. Such graphs can be visually very impressive, and are often useful in areas where a visual interpretation of data is required.

Applications for this sort of graph range from the oil exploration industry, where they are used for the display and interpretation of sonar pictures of the earth's crust, to plots of the time and frequency characteristics of loudspeaker performance, where there are so many subjective variables that a straight 'rational' approach often isn't enough.

Figure 2–6 is an example generated by another mathematical function incorporated in the program THREE-D CURVE.

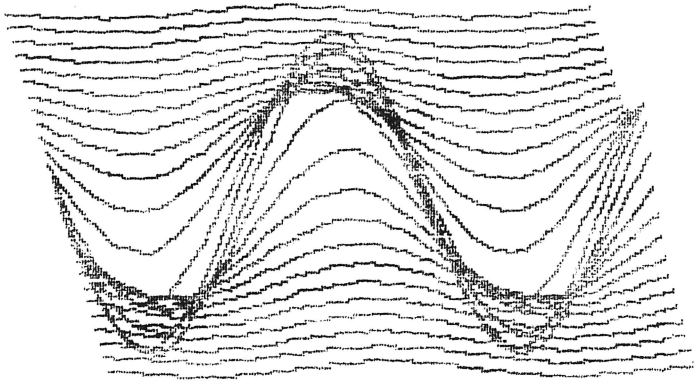


Figure 2–6

THREE-D CURVE

```
10  HGR
20  HCOLOR= 3
30  FOR Z =  - 10 TO 10 STEP 0.75
40 X = 0
50  GOSUB 1000
60  HPLOT X * 22 + Z * 2 + 30,Z * 6 +
Y * 7 + 70
70  FOR X = 0 TO 10 STEP 0.2
80  GOSUB 1000
90  HPLOT  TO X * 22 + Z * 2 + 30,
Z * 6 + Y * 7 + 70
100  NEXT X
110  NEXT Z
120  STOP
1000 Y =  SIN (X) * 30 / (Z ^ 2 + 4)
1010  RETURN
```

The function represented in Figure 2–6 has three variables: X (horizontal), Y (vertical) and Z (into the page). The graph is given by the equation

$$Y = \frac{\sin(X) \cdot 30}{Z^2 + 4}$$

Most of the program should be fairly self-explanatory, with the possible exception of line 90. This is similar to the plotting equations used before, except that a change in Z will cause a change in both the distance from the top of the screen and the distance from the 'back' of the screen. This is because we have 'turned' the graph slightly so that the edge can be seen. If it was just straight on, we would not be able to see the various separate traces as Z changes.

Another interesting plotting problem is drawing a circle on the screen. This is mathematically more complicated than drawing graphs, but easier in terms of its programming.

CIRCLE

```
10 PI = 3.1415926
20  HGR
30  HCOLOR= 3
40 XC = 140:YC = 80
50 R = 70
60 XR = 0:YR = R
70  HPLOT XC + XR,YC + YR
80  FOR TH = 0 TO 2 * PI STEP 0.01
90 XR = R *  SIN (TH)
```

```
100 YR = R *  COS (TH)
110  HPLOT  TO XC + XR,YC + YR
120  NEXT TH
```

The first step (line 10) is to feed into the computer a value for pi.

Line 20 clears the top part of the screen, and line 30 sets the color.

Line 40 sets the co-ordinates of the centre of the circle. Line 50 sets the radius.

Lines 60 and 70 plot the first point on the circumference.

Lines 80 and 120 step TH (short for theta, the angle around the circle) from 0 to 2 pi. At each point, lines 90 and 100 calculate the horizontal and vertical displacements from the centre.

Line 110 plots a line to the new point.

This program draws a large circle in the middle of the screen (the example shown in Figure 2–7 has been distorted by hardware).

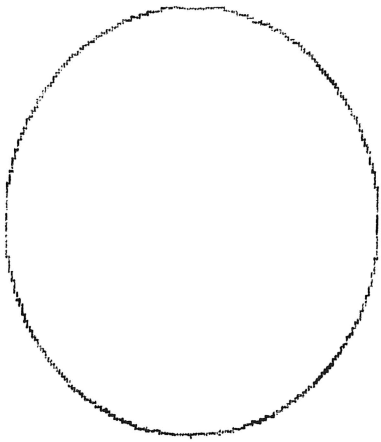


Figure 2–7

The Arts

We have seen that the Apple can be used for the display of business and scientific information, and also for generating mathematical figures for the purposes of education. Another, less well-known use for computer graphics is in the production of 'computer art' which generally takes the form of geometric abstracts. Some artists have explored the possibilities of 'computer art' in great depth.

The programs listed here are intended as 'idea starters'. They are short and fairly simple. You could try these out, and then modify them, and experiment with your own ideas. Try, for example, the addition of color. The last program in this sequence (PATTERN 4) might even gain something by the addition of sound! Figures 2–8 to 2–11 are samples of the graphics generated by these programs.

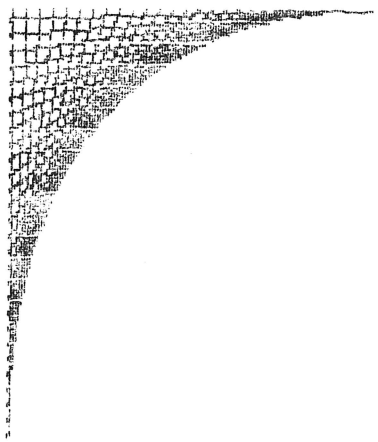


Figure 2–8

PATTERN 1

```
10  HGR
20  HCOLOR= 3
30  FOR I = 1 TO 140 STEP 5
40  HPLOT I + 10,10
50  HPLOT  TO 10,150 - I
60  NEXT I
```

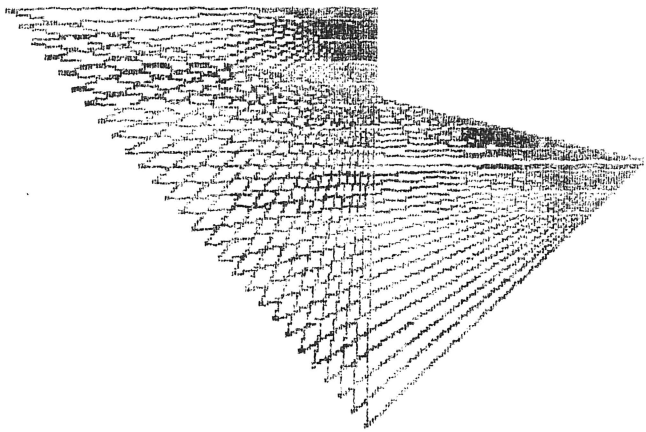


Figure 2-9

PATTERN 2

```
10  HGR
20  HCOLOR= 3
30  FOR I = 1 TO 140 STEP 5
40  HPLOT 140,10
50  HPLOT  TO I,I + 10
60  HPLOT  TO 240,60
70  NEXT I
```

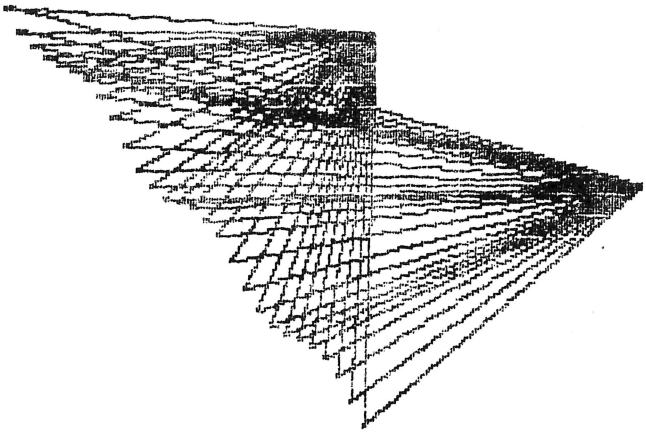


Figure 2–10

PATTERN 3

```
10  HGR
20  HCOLOR= 3
30  FOR I = 1 TO 140 STEP 5
40  HPLOT 140,10
50  HPLOT  TO I,I + 5 *  SIN (I / 7)
60  HPLOT  TO 240,60
70  NEXT I
```

Figure 2–11

PATTERN 4

```
10  HGR
20 X = 40:Y = 40
30  HCOLOR= 3
40  IF  RND (1) > 0.5 THEN  HCOLOR= 0
50  HPLOT X,Y
60 X = X + ( RND (1) - 0.5) * 20
70 Y = Y + ( RND (1) - 0.5) * 20
80  IF X > Y THEN X = Y
90  IF Y > 140 THEN Y = 140
100  IF X < 1 THEN X = 1
110  IF Y < 1 THEN Y = 1
120  HPLOT  TO X,Y
130  GOTO 30
```

Rotation

Now we come to the techniques that we'll be using throughout the rest of the book. In order to store and manipulate a shape that has been fed into the computer, we will use a series of 'co-ordinates' (i.e. pairs of numbers) that will 'spell out' the shape on the screen. The following program will draw a shape (in this case, an arrow) on the screen.

ARROW

```
10  DIM A(8,2)
20  HCOLOR= 3
30  FOR I = 1 TO 8
40  READ A(I,1),A(I,2)
50  NEXT I
60  DATA  140,80,140,58,135,58,143,49
70  DATA  151,58,147,58,147,80,140,80
80  REM  NOW PLOT THE RESULT
90  HGR
100  HPLOT A(1,1),A(1,2)
110  FOR I = 2 TO 8
120  HPLOT  TO A(I,1),A(I,2)
130  NEXT I
```

The first part of the program (up to line 80) simply creates the array A(,), and puts the values of the co-ordinates into it. The rest of the program uses these co-ordinates to plot the shape onto the screen. Running this program gives the result shown in Figure 2–12.

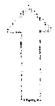


Figure 2–12

This technique enables us to move the shape around, alter it and even rotate it using some mathematics.

For example, if we insert this loop into the program,

```
72 FOR I=1 TO 8
74 A(I,1)=A(I,1)/2
76 NEXT I
```

The result will be as shown in Figure 2–13.

Figure 2–13

And if we then make line 74:

```
74 A(I,2)=A(I,2)/2
```

The result will be as shown in Figure 2–14.

Figure 2–14

Rotation

It is possible to rotate the shape, too. The following program rotates the shape and allows you to change its size.

ROTATE AND SCALE

```
10  DIM A(8,2)
20  HCOLOR= 3
30  FOR I = 1 TO 8
40  READ A(I,1),A(I,2)
50  NEXT I
60  DATA  140,80,140,58,135,58,143,49
70  DATA  151,58,147,58,147,80,140,80
80  REM  NOW PLOT THE RESULT
90  HGR
100  HPLOT A(1,1),A(1,2)
110  FOR I = 2 TO 8
120  HPLOT  TO A(I,1),A(I,2)
130  NEXT I
140  REM  FIND THE MIDDLE
150 XL = A(1,1):XH = XL
160 YL = A(1,2):YH = YL
170  FOR I = 1 TO 8
180  IF A(I,1) > XH THEN XH = A(I,1)
190  IF A(I,1) < XL THEN XL = A(I,1)
200  IF A(I,2) > YH THEN YH = A(I,2)
210  IF A(I,2) < YL THEN YL = A(I,2)
220  NEXT I
230 XM = (XH + XL) / 2
240 YM = (YH + YL) / 2
250  HPLOT XM,YM
260  INPUT "ROTATION ";R
270  INPUT "SCALE ";S
280  FOR I = 1 TO 8
```

```
290 X = A(I,1) - XM:Y = A(I,2) - YM
300  REM  FIRST, THE SCALING
310 X = X * S
320 Y = Y * S
330  REM  NEXT, THE ROTATION
340  IF X = 0 THEN X = 1E - 7
350 TH =  ATN (Y / X)
360  IF X < 0 THEN TH = TH + 3.1415926
370 TH = TH + R
380 RD =  SQR (X * X + Y * Y)
390 X = RD *  COS (TH)
400 Y = RD *  SIN (TH)
410 A(I,1) = XM + X
420 A(I,2) = YM + Y
430  NEXT I
440  GOTO 80
```

The first part is exactly the same as the ARROW program.

Lines 140 to 250 find and plot the centre of the figure. This is done by finding the maximum and minimum values of each of the co-ordinates (in other words the extreme right, left, top and bottom edges of the figure), and then taking a point halfway between them.

A maximum and a minimum value for X (distance from the left of the screen) and for Y (distance from the top of the screen), are set and held in variables XH, YH, XL and YL (for X and Y, high and low). The loop at lines 170 to 220 goes through each of the values in the figure in turn, and compares the current maximum and minimum values with each point. When it finds that one of the X values (which are stored in A(1,)) is greater than XH, for example, it will make XH equal to that higher value. In this way, by the time we get to line 230, XH, YH, XL and YL are the required highest and lowest values.

Lines 230 and 240 find the co-ordinates XM and YM of the middle of the figure. Line 250 puts a dot there.

Lines 260 and 270 take in values for the required rotation and scale for the figure.

A scale of 1 will leave the size of the figure unaltered, while a scale of 0.5 will halve the size of the figure, a scale of 2 double it, and so on.

A rotation of 0 will leave the figure unrotated, while a rotation of 3.14 will rotate it half way round (we are working with 'radians' here — there are roughly 3.14 radians in 180 degrees, or half a rotation).

The loop from 280 to 430 changes the values of the figure's co-ordinates according to the given rotation and scale and line 440 re-starts the program.

Line 290 finds X and Y, which are the distances horizontally and vertically from the middle of the figure (XM, YM) to the point we're working with. At lines 410 and 420, we will find the *new* value of A(I,1) and A(I,2) by adding X and Y to the values of XM and YM.

Lines 310 and 320 deal with scaling. If S is the required scale, we can change the scale of the figure by simply multiplying X and Y by the value of S. In effect this will increase or decrease the distance of each point from the centre by an appropriate factor. If, for example, the scale required is 2, the X and Y values of every point on the figure are doubled. This moves each point to twice the initial distance from the middle. The figure will end up twice the original size.

Line 340 makes sure that we never try to divide by 0 in line 350. If we *did* try to divide by 0, the computer would stop and tell us that something is wrong. When X is 0, line 340 will change it to a very small number so close

to 0 that we won't be able to tell the difference on the screen.

Line 350 finds TH, which is the angle (in radians) of a given point on the figure relative to the middle. A point exactly to the right of the middle (no matter what distance) has a TH of 0. A point below that one would have a TH of more than 0, and so on round a complete circle. A point just *above* the one with TH=0 would have a TH of about 6.28 (6.28 radians = 360 degrees).

Mathematically-minded readers may be wondering why TH runs contrary to convention (which says that the angle increases as you go anti-clockwise). It is so for convenience since Y increases *down* the screen in the Apple screen display.

Line 360 deals with cases where both X and Y are below zero.

Line 370 is the line that actually does the rotation. It adds the value of the required rotation, R, onto TH, thus rotating each point on the figure clockwise by R radians.

The new values of X and Y are calculated using the new value of TH. Line 380 finds the new distance from the point on the figure to the middle. It is an application of that well-known mathematical law which says 'The square of the hypotenuse of a triangle is equal to the sum of the squares of the other two sides'. Imagine that the distances X and Y lie along the bottom and right-hand sides of a right-angled triangle. Line 380 finds the length of the hypotenuse which represents the distance from the point under consideration to the middle.

Having found the new angle, TH, and the distance from the middle, RD, it is simple to calculate the new values of X and Y. Lines 390 and 400 do this.

Sample Run

Figure 2–15 is an example of a result obtained by running the program.

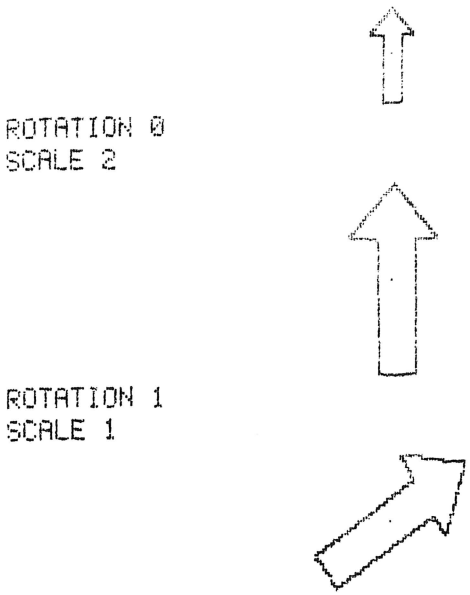


Figure 2–15

The program does have its limitations though. If you use too large a 'scale' factor, the figure will become too large to fit on the screen. The Apple will give an error when you tell it to plot a point beyond the edge of the screen. But we can't ignore such points because a line between two points that are both off the screen may cross

the screen. It is necessary to determine which segments of such lines do actually fall within the screen.

We can write a subroutine to deal with this problem, but first we need to decide what we want this routine to do (it may sound fairly obvious, but it's a step that many people leave out to their cost when writing small routines!). We want to deal with the case where one or both of the ends of a line lie outside the screen area, and find the part of that line that lies inside the area.

Let us imagine that the line segment of interest is part of an infinite line. An infinitely long line (i.e. one that stretches away in both directions off the edge of the page) will cross the edges of the screen either twice (if the line crosses the screen) or no times (if it doesn't cross the screen). We can find the equation that describes this infinite line, and then find the places where it will cross the four lines that mark out the screen boundary.

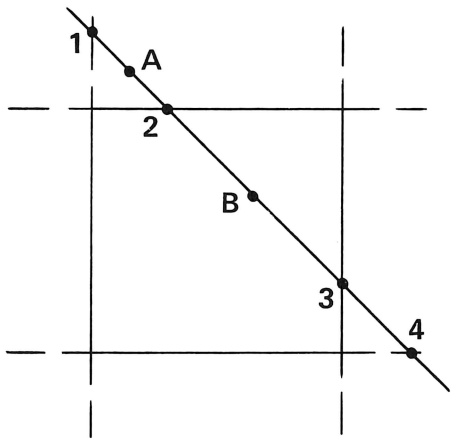


Figure 2–16

In Figure 2–16 the box represents the screen area. The line we are interested in is from point A to point B, and is a part of the infinite line.

There are two places where this extended line crosses the screen borders (points 2 and 3). These are the two points we need to find. Having found them, it is a relatively simple matter to figure out whether to draw the line from 2 to B, or from 2 to 3, or from B to 3.

In order to find these two crossing points (2 and 3), we find each of the crossing points in turn (1, 2, 3 and 4) and check to see whether each is on the border of the box. Finding these four points turns out to be easy once we have the equation of the line. The four sides of the box are X=1, X=279, Y=1 and Y=191 in terms of screen co-ordinates. The crossing points are calculated by substituting these values of X and Y in the equation of the line.

The next step is to check each of the crossing points to see if it lies on the side of the box, and finally find out which two of our points to draw the line between.

The subroutine EXPLOT deals with cases where parts of a figure extend beyond the screen after scaling or rotation.

EXPLOT

```
1  DIM P(2),Q(2):X9 = 279:Y9 = 191
2 P1 = 0.1:N1 = 1:N2 = 2
6000  REM   ENTRY TO ROUTINE
6010  IF F = 0 THEN OX = X:OY = Y:
RETURN : REM  FIRST POINT IN FIGURE
6020  IF OX > X THEN  GOTO 6050
```

```
6030 X1 = OX:Y1 = OY:X2 = X:Y2 = Y
6040  GOTO 6060
6050 X1 = X:Y1 = Y:X2 = OX:Y2 = OY
6060 OX = X:OY = Y
6065  IF X1 > 1 AND X1 < X9 AND X2 > 1
AND X2 < X9 AND Y1 > 1 AND Y1 < Y9 AND
Y2 > 1 AND Y2 < Y9 THEN  HPLOT X1,Y1 TO
X2,Y2: RETURN
6070  IF Y2 = Y1 THEN Y2 = Y1 + P1
6080  IF X2 = X1 THEN X2 = X1 + P1
6090 G = (Y2 - Y1) / (X2 - X1)
6100 X0 = X1 - Y1 / G
6110 N = 0
6120 Y = G * (1 - X0)
6130  IF Y < N1 OR Y > Y9 THEN
GOTO 6150
6140 N = N + N1:P(N) = N1:Q(N) = Y
6150 Y = G * (X9 - X0)
6160  IF Y < N1 OR Y > Y9 THEN
GOTO 6180
6170 N = N + N1:P(N) = X9:Q(N) = Y
6180  IF N = N2 THEN  GOTO 6290
6190  IF N = N1 AND Q(N) = N1 THEN
GOTO 6230
6200 X = X0 + N1 / G
6210  IF X < N1 OR X > X9 THEN
GOTO 6230
6220 N = N + N1:P(N) = X:Q(N) = N1
6230  IF N = N2 THEN  GOTO 6290
6240  IF N = 1 AND Q(N) = Y9 THEN
GOTO 6280
6250 X = X0 + Y9 / G
6260  IF X < N1 OR X > X9 THEN
GOTO 6280
6270 N = N + N1:P(N) = X:Q(N) = Y9
6280  IF N < N2 THEN  RETURN
6290 N = N2
6300  IF P(N1) < P(N2) THEN N = N1
6310 NN = 3 - N
6320  IF X1 > P(NN) OR X2 < P(N) THEN
RETURN
```

```
6330  IF X1 >  = P(N) THEN P(N)
= X1:Q(N) = Y1
6340  IF X2 <  = P(NN) THEN P(NN)
= X2:Q(NN) = Y2
6350  HPLOT P(N),Q(N) TO P(NN),Q(NN)
6360  RETURN
```

This subroutine fits into the main program by putting lines 1 and 2 at the start — they define the arrays needed, and set up the constants.

Line 1 is pretty straightforward. P() and Q() are the arrays that are going to hold the two points to plot between. X9 and Y9 are the right-hand and bottom edges of the screen.

Line 2 sets some numeric constants to speed up the operation of the routine.

The Apple works faster with variables than it does with numbers. For example A=A+N1 will be processed slightly faster than A=A+1.

In the subroutine P1 is 'point one', N1 is 'number one' and N2 is 'number 2'.

On entry to the routine (at line 6000), variables X and Y will have been set by the main program to the screen position to which a line is to be plotted. The routine will retain in variables OX and OY (old X and old Y) the co-ordinates of the last position.

If you wish, you can make this the first point of a figure by setting F=0. F stands for 'first point'. Line 6010 deals with this situation — no plotting takes place, and all the routine does is to set OX and OY to this 'last point plotted'.

Lines 6020 to 6060 set up X1, Y1, X2 and Y2. These are the two points that the line runs between. The reason why X, Y, OX and OY are not used here is that the

program makes sure that X1 is *less than* X2; that is X1, Y1 is the *left-most* point.

After X1, Y1, X2 and Y2 have been set up, line 6060 sets OX and OY so that they will be ready for the next time the routine is called.

In order to speed up the routine, line 6065 quickly takes care of the case where both points are on the screen.

Lines 6070 and 6080 deal with the case where the line is either vertical or horizontal as this would confound the maths (by causing division by 0 and other 'nasties') and so we add 0.1 to X2 or Y2 which has no significant effect on the line as it appears on the screen.

Gradient

Line 6090 finds the 'gradient' or slope of the line.

Line 6100 finds the 'Y-intersection'; this is the value of X when Y=0 and is held in the variable X0.

Knowing G (the gradient) and X0, we know enough to accurately describe the line, and we can write its equation as Y=G(X−X0).

Lines 6120 to 6280 find the four places where the 'extended' line crosses the right, left, top and bottom boundaries of the screen.

The left boundary is the line where X=1. Firstly, we need to find out where the diagonal line crosses this X=1 line, and secondly whether the crossing point is on the border of the box.

Line 6120 finds the crossing point by using the equation of the diagonal line (as shown above) and

substituting 1 for X, i.e. Y=G(1−X0). This substitution gives us (since we already know G and X0) the value of Y on the X=1 line where the diagonal line crosses.

Line 6130 determines whether the Y-value of the crossing point is within the range of values corresponding to the screen border. If the point does not lie on the edge of the screen, we go to line 6150 to try the next crossing point. If Y *is* within the limits, then we go on to line 6140.

Line 6140 increases the value of N by 1 (which is counting how many points we have come across) and stores the co-ordinates of the last point in the P() and Q() arrays for later use.

Now we only need *two* valid points for our system to work. It is possible (if the diagonal line passes right through the *corner* of the box) to get more than two coming up, and so we check at line 6180 to see how many we have, and skip the next part if there are two already.

The equations at lines 6120 and 6150 test the left and right edges of the box respectively, while the equations at lines 6200 and 6250 test the top and bottom of the box.

Lines 6190 and 6240 deal with the situation where the line passes through *both* corners of the box.

When we reach line 6280, we should have either two points (i.e. N=2) or none (N=0) (none if the diagonal line does not cross the box at all). Line 6280 RETURNs to the main program if there are no crossing points on the border of the box.

The remainder of the routine (lines 6290 to 6340) finds out which of the two crossing points (held in P() and

Q()) is the left-most, and which two points the line is to be drawn between.

Line 6350 plots the line on the screen.

Here is a listing of the ROTATE AND SCALE program with the subroutine included, followed by a sample run (see Figure 2–17).

ROTATE SCALE AND EXPLOT

```
1  DIM P(2),Q(2):X9 = 279:Y9 = 191
2 P1 = 0.1:N1 = 1:N2 = 2
10  DIM A(8,2)
20  HCOLOR= 3
30  FOR I = 1 TO 8
40  READ A(I,1),A(I,2)
50  NEXT I
60  DATA  140,80,140,58,135,58,143,49
70  DATA  151,58,147,58,147,80,140,80
80  REM  NOW PLOT THE RESULT
90  HGR
95 F = 0
100 X = A(1,1):Y = A(1,2): GOSUB 6000
105 F = 1
110  FOR I = 2 TO 8
120 X = A(I,1):Y = A(I,2): GOSUB 6000
130  NEXT I
140  REM  FIND THE MIDDLE
150 XL = A(1,1):XH = XL
160 YL = A(1,2):YH = YL
170  FOR I = 1 TO 8
180  IF A(I,1) > XH THEN XH = A(I,1)
190  IF A(I,1) < XL THEN XL = A(I,1)
200  IF A(I,2) > YH THEN YH = A(I,2)
210  IF A(I,2) < YL THEN YL = A(I,2)
220  NEXT I
```

```
230 XM = (XH + XL) / 2
240 YM = (YH + YL) / 2
250  HPLOT XM,YM
260  INPUT "ROTATION ";R
270  INPUT "SCALE ";S
280  FOR I = 1 TO 8
290 X = A(I,1) - XM:Y = A(I,2) - YM
300  REM  FIRST, THE SCALING
310 X = X * S
320 Y = Y * S
330  REM  NEXT, THE ROTATION
340  IF X = 0 THEN X = 1E - 7
350 TH =  ATN (Y / X)
360  IF X < 0 THEN TH = TH + 3.1415926
370 TH = TH + R
380 RD =  SQR (X * X + Y * Y)
390 X = RD *  COS (TH)
400 Y = RD *  SIN (TH)
410 A(I,1) = XM + X
420 A(I,2) = YM + Y
430  NEXT I
440  GOTO 80
6000  REM   ENTRY TO ROUTINE
6010  IF F = 0 THEN OX = X:OY = Y:
RETURN : REM  FIRST POINT IN FIGURE
6020  IF OX > X THEN  GOTO 6050
6030 X1 = OX:Y1 = OY:X2 = X:Y2 = Y
6040  GOTO 6060
6050 X1 = X:Y1 = Y:X2 = OX:Y2 = OY
6060 OX = X:OY = Y
6065  IF X1 > 1 AND X1 < X9 AND X2 > 1
AND X2 < X9 AND Y1 > 1 AND Y1 < Y9 AND
Y2 > 1 AND Y2 < Y9 THEN  HPLOT X1,Y1 TO
X2,Y2: RETURN
6070  IF Y2 = Y1 THEN Y2 = Y1 + P1
6080  IF X2 = X1 THEN X2 = X1 + P1
6090 G = (Y2 - Y1) / (X2 - X1)
6100 X0 = X1 - Y1 / G
6110 N = 0
6120 Y = G * (1 - X0)
```

```
6130  IF Y < N1 OR Y > Y9 THEN
GOTO 6150
6140 N = N + N1:P(N) = N1:Q(N) = Y
6150 Y = G * (X9 - X0)
6160  IF Y < N1 OR Y > Y9 THEN
GOTO 6180
6170 N = N + N1:P(N) = X9:Q(N) = Y
6180  IF N = N2 THEN  GOTO 6290
6190  IF N = N1 AND Q(N) = N1 THEN
GOTO 6230
6200 X = X0 + N1 / G
6210  IF X < N1 OR X > X9 THEN
GOTO 6230
6220 N = N + N1:P(N) = X:Q(N) = N1
6230  IF N = N2 THEN  GOTO 6290
6240  IF N = 1 AND Q(N) = Y9 THEN
GOTO 6280
6250 X = X0 + Y9 / G
6260  IF X < N1 OR X > X9 THEN
GOTO 6280
6270 N = N + N1:P(N) = X:Q(N) = Y9
6280  IF N < N2 THEN  RETURN
6290 N = N2
6300  IF P(N1) < P(N2) THEN N = N1
6310 NN = 3 - N
6320  IF X1 > P(NN) OR X2 < P(N) THEN
RETURN
6330  IF X1 >  = P(N) THEN P(N)
= X1:Q(N) = Y1
6340  IF X2 <  = P(NN) THEN P(NN)
= X2:Q(NN) = Y2
6350  HPLOT P(N),Q(N) TO P(NN),Q(NN)
6360  RETURN
```

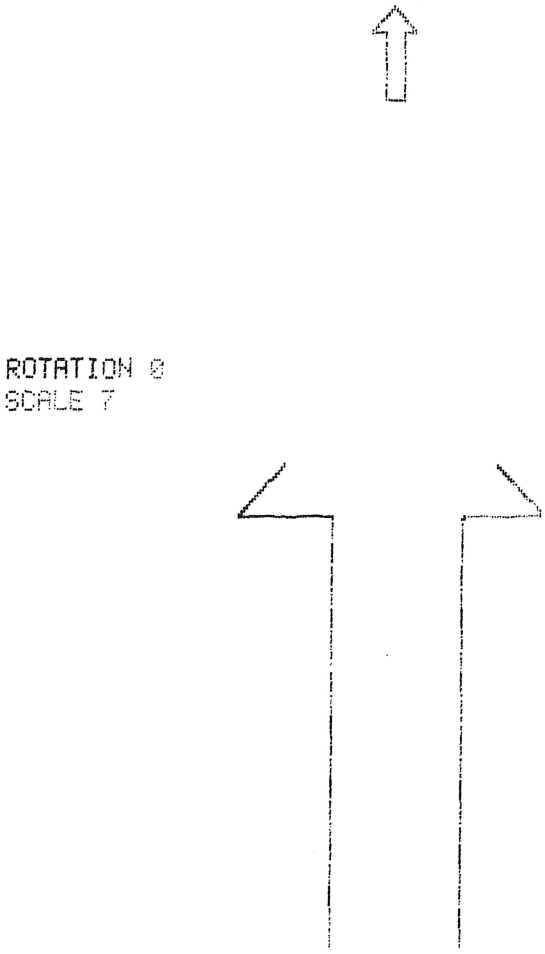


Figure 2-17

The MAKE FIGURE Program

One of the most intriguing possibilities is the use of the computer to automate 'flat' animation. 'Flat' animation refers to transforming one shape into another. 'Three-dimensional' animation means dealing with 'solid' shapes stored as images inside the computer.

In this section we develop a program that allows us to create a 'figure' on the screen (as opposed to a 'shape', which we dealt with earlier in the book). The program will allow us to alter the figure, store it for later reference, and so on. Using the program, we will generate two figures, stored on tape, disc or paper, which have the same number of points. If you like, one figure is a variation on the other. For example, we might design one figure which is a drawing of a mouth without a grin, then alter that figure so that it is a drawing of the same mouth, but grinning. Then we can feed both figures into a *second* program which *automatically* generates a series of figures on the screen which go from the first (unsmiling) figure to the second (smiling) figure in a series of gradual steps.

The application of this technique will be obvious to anyone who has been involved in animation. What normally happens in a situation like this is that all of the shapes in between have to be drawn by hand one at a time, and then filmed. However, using the computer we can generate the sequence of figures automatically. These can then form the basis of the series of drawings thus saving time and producing a smoother result.

The MAKE FIGURE program allows us to create, store, recall and modify shapes. Just to make it a little more flexible, a choice of color is available.

Using MAKE FIGURE

When you first RUN the program, the screen will blank, and the following information will appear at the bottom:

```
U=UP M=DOWN K=RIGHT H=LEFT J=FIX
O=ON/OFF C=COLOUR A=ADD
E=ERASE F=FINISH N=NEXT P=PREVIOUS
ON
```

This tells you what each of the keys will do. In the top part of the screen is a little square. Pressing the "U" key (without touching the RETURN key — it's not needed) will make the little square (called the 'cursor') move up, and so on. (It may look at first glance like the allocation of keys is a little arbitrary — but take a closer look at the way the keys are laid out.) Notice that the cursor moves at a reasonable speed if you hold down the REPT (repeat) key while holding down one of these 'cursor movement keys'.

Pressing the O (the *letter* O) key, the word 'ON' at the bottom of the screen will change to OFF, and back again the next time you press O. Note that fact — you'll need it later.

Now press the C key — a message will appear telling you:

```
COLOUR IS 3 — NEW COLOUR?
```

Enter 3 and press RETURN. The colors are allocated according to the Apple's color scheme, which is explained earlier in this book.

Make sure that the message at the bottom of the screen reads 'ON' (press the O key if it doesn't), and then using the REPT, U, M, K and H keys, move the cursor to somewhere in the top right hand corner of the screen. Press the C key, and enter 3 to make sure that the color is set correctly. Now press the A key to add a point to the figure. The program will draw a line in the current color (which is 3 = white) from the last point (the middle of the screen) to the new point. Now move the cursor down a little, and press the J key. The whole thing should disappear, and then reappear with the point that you've just added slightly lower down. You can move the point by pressing the J key.

By moving the cursor around the screen and pressing the A key, make a figure which contains a half dozen points. Now press the P key. The cursor should jump to the last point you put in — the *'previous'* point. Press the P key a few more times, then press the N key. This will move the cursor in the other direction (N = next). By pressing the P and N keys, you can move the cursor to any of the points on the figure (except the very first one).

Now move the cursor to one of the points using the P and N keys, then move the cursor somewhere else using only the REPT, U, M, K and H keys. Press the J key. This causes the program to move the point that you have just chosen (using the P and N keys) to the position that you've just chosen (using the U, M, K and H keys). The figure will be re-drawn with the change made.

Now press the E key. The figure will be re-drawn with the last point missing. E erases the last point. Erase all of the points in the figure using the E key. Now put a single line on the screen by moving the cursor to the top right-hand part of the screen and pressing the A key.

Now press C, and change the color from 3 to 2.

Without moving the cursor, press the J key. The point will not move, but the color of the line will change from white to blue-green. (On black and white screens, it will change from solid to dotted.) You can change the color of a line by changing the current color using the C key, and then by 'FIXing' the point using the J key.

Put a few more lines onto the screen using color = 3. Change the color to 0 (black), and draw a line that crosses some of the previous lines. Notice that where the lines cross, the white lines have a black spot where the new black line crosses. To avoid this, we need a way of stopping the series of lines at one part of the screen and re-starting it at another, without drawing black lines. By pressing the O key you can change the message at the bottom of the screen from ON to OFF and back again. When the message reads OFF, the computer is not drawing lines. If you like, we have changed the current color to 'invisible'.

In this way we can draw a shape at one side of the screen, then turn the line OFF, move the cursor to the other side of the screen, press A to draw an 'invisible' line, set it back to ON, and then continue with the next part of the shape.

Having created and modified a figure we need a way of storing and recalling it.

Store and recall

Press the F key. That tells the machine that you are finished with the current figure. The program will ask you whether you want to:

S=SAVE THIS FIGURE, R=READ A NEW FIGURE ?

At this stage, you will not have any shapes saved, so press S. Later, when you have some figures stored, you can enter R to erase the current figure and read in the new one. If you want to start the program up and read a new figure in immediately, you can just press F when the first message comes up. If you enter either S or R, the program will lead you through a series of questions (i.e. disc, tape or screen), which will allow you to use any one of these three methods for storing or recalling a figure.

In order to animate a figure, we first create the starting shape (for example, Figure 2–18).

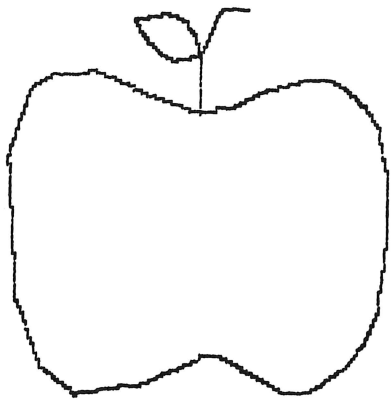


Figure 2–18

We save it to disc, or onto tape, or onto the screen and from there to paper by hand. Having done that, we alter the starting figure until we obtain the final figure. This must be done without changing the number of points, by adding or erasing any of them. See Figure 2–19.

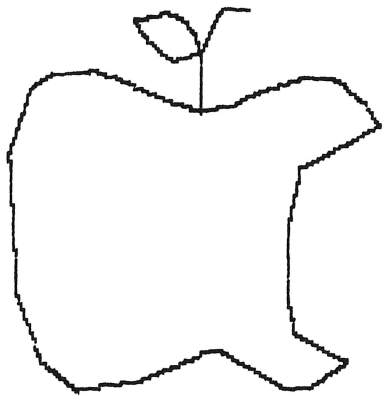


Figure 2–19

Then we save this final version of the shape. Later in this section, there is a listing of the programs which can be used to generate a series of figures based on these two which will provide an animated motion between the two (see Figure 2–20).

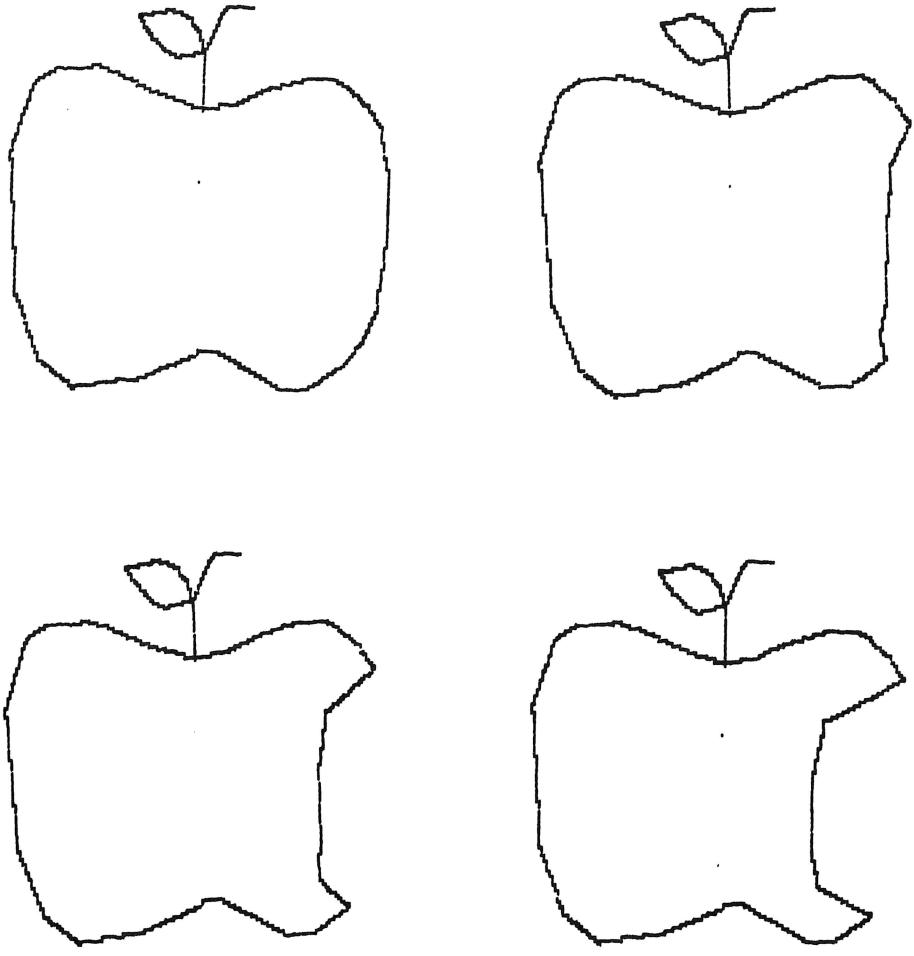


Figure 2–20

MAKE FIGURE

```
10  DIM C(100,2),HC%(100),DM%(101)
20  GOSUB 4000: REM  INITIALISE SHAPE
TABLE, ETC
30 X = 140:Y = 80
40  XDRAW 1 AT X,Y
50 P = 1: REM  NEXT AVAILABLE POSITION
60 CP = 0: REM  CURRENT POSITION
70 C(0,1) = X:C(0,2) = Y:DM%(0) = 1
80 O = 1: REM O=1 IS 'ON', O=0 IS 'OFF'
90  GOSUB 530
100 CL = 3: REM  HOLDS CURRENT COLOUR
110  REM
120  REM  NEXT KEY
130  GET A$
140  IF A$ = "U" THEN Y = Y - 1
150  IF A$ = "M" THEN Y = Y + 1
160  IF A$ = "K" THEN X = X + 1
170  IF A$ = "H" THEN X = X - 1
180  IF X > 279 THEN X = 279
190  IF X < 0 THEN X = 0
200  IF Y > 159 THEN Y = 159
210  IF Y < 0 THEN Y = 0
220  IF A$ = "P" THEN  GOSUB 910
230  IF A$ = "N" THEN  GOSUB 1000
240  IF A$ = "E" THEN  GOSUB 740
250  IF A$ = "C" THEN  GOSUB 440
260  IF A$ = "F" THEN  GOTO 1320
270  IF A$ = "A" THEN  GOSUB 600
280  IF A$ = "J" THEN  GOSUB 1080
290  IF A$ = "O" THEN  GOSUB 370
300  IF A$ <  > "U" AND A$ <  > "M" AND
A$ <  > "K" AND A$ <  > "H" THEN
GOTO 130
310  REM
320  REM  NOW MOVE POINTER
330  XDRAW 1
340  XDRAW 1 AT X,Y
350  GOTO 130
360  REM
```

```
370  REM  CHANGE STATE OF 'O'
380  IF O = 1 THEN O = 0: GOTO 400
390 O = 1
400  GOSUB 530
410  RETURN
420  REM
430  REM  CHANGE COLOUR
440  PRINT : PRINT : PRINT : PRINT
450  PRINT "COLOUR IS ";CL;" - NEW
COLOUR ";
460  INPUT A
470  IF A < 0 OR A > 7 THEN
PRINT "CAN'T HAVE ";A: GOTO 450
480 CL = A
490  GOSUB 530
500  RETURN
510  REM
520  REM  PRINT MNEMONICS
530  PRINT : PRINT : PRINT "U=UP M=DOWN
K=RIGHT H=LEFT J=FIX"
540  PRINT "O=ON/OFF C=COLOUR A=ADD"
550  PRINT "E=ERASE F=FINISH N=NEXT
P=PREVIOUS"
560  IF O = 1 THEN  PRINT "ON";
570  IF O = 0 THEN  PRINT "OFF";
580  RETURN
590  REM
600  REM  ADD A POINT
610 IF P > 100 THEN  RETURN
620 IF C(P - 1,1) = X AND C(P - 1,2) = Y
THEN  RETURN : REM  ONE FIX PER POSITION
ONLY
630 C(P,1) = X:C(P,2) = Y
640 HC%(P) = CL
650 DM%(P) = O
660  IF O = 0 THEN  HPLOT X,Y: GOTO 700
670  HCOLOR= CL
680  HPLOT C(P - 1,1),C(P - 1,2) TO X,Y
690  HCOLOR= 3
700 P = P + 1:CP = P - 1
710  RETURN
```

```
720  REM
730  REM  ERASE LAST ENTRY
740 P = P - 1: IF P = 0 THEN P = 1
750  HGR
760  VTAB 24
770  IF P = 1 THEN  GOTO 840
780  FOR I = 1 TO P - 1
790  HCOLOR= HC%(I)
800  IF DM%(I) = 0 THEN  HPLOT
C(I,1),C(I,2): GOTO 820
810  HPLOT C(I - 1,1),C(I - 1,2) TO
C(I,1),C(I,2)
820  NEXT I
830  HCOLOR= 3
840 CP = P - 1
850 X = C(CP,1):Y = C(CP,2)
860  XDRAW 1 AT X,Y
870  GOSUB 530
880  RETURN
890  REM
900  REM  PREVIOUS POINT
910  XDRAW 1
920 CP = CP - 1
930  IF CP < 1 THEN CP = 1
940 X = C(CP,1):Y = C(CP,2)
950 O = 1 - DM%(CP): GOSUB 370
960  XDRAW 1 AT X,Y
970  RETURN
980  REM
990  REM  NEXT POINT
1000  XDRAW 1
1010 CP = CP + 1
1020  IF CP > P - 1 THEN CP = P - 1
1030 X = C(CP,1):Y = C(CP,2)
1040 O = 1 - DM%(CP): GOSUB 370
1050  XDRAW 1 AT X,Y
1060  RETURN
1070  REM
1080  REM  FIX A POINT
1090  IF CP = 0 THEN  RETURN
1100  REM  ERASE POINTER
```

```
1110  XDRAW 1
1120  REM  ERASE LINES
1130  HCOLOR= 0
1140  HPLOT C(CP - 1,1),C(CP - 1,2) TO
C(CP,1),C(CP,2)
1150  IF CP = P - 1 THEN  GOTO 1170
1160  HPLOT C(CP,1),C(CP,2)
TO C(CP + 1,1), C(CP + 1,2)
1170  REM  CHANGE VALUE
1180 DM%(CP) = O:HC%(CP) = CL
1190 C(CP,1) = X:C(CP,2) = Y
1200  IF O = 0 THEN  HPLOT X,Y:
GOTO 1230
1210  HCOLOR= CL
1220 HPLOT C(CP - 1,1),C(CP - 1,2)
TO X,Y
1230  IF CP = P - 1 THEN  GOTO 1270
1240  IF DM%(CP + 1) = 0 THEN  HPLOT
C(CP + 1,1),C(CP + 1,2): GOTO 1270
1250  HCOLOR= HC%(CP + 1)
1260 HPLOT X,Y TO C(CP + 1,1),
C(CP + 1,2)
1270  HCOLOR= 3
1280  XDRAW 1 AT X,Y
1290  RETURN
1300  REM
1310  REM  FINISHED WITH THIS FIGURE
1320  PRINT : PRINT : PRINT : PRINT
1330  PRINT "S=SAVE THIS FIGURE, R=READ
NEW FIGURE"
1340  INPUT A$
1350  IF A$ = "S" THEN  GOTO 1400
1360  IF A$ = "R" THEN  GOTO 1810
1370  GOTO 1320
1380  REM
1390  REM  SAVE SHAPE
1400  PRINT : PRINT : PRINT : PRINT
1410  PRINT "D=DISC T=TAPE S=SCREEN ";
1420  INPUT A$
1430  IF A$ = "D" THEN  GOTO 1680
1440  IF A$ = "T" THEN  GOTO 1580
```

```
1450  IF A$ = "S" THEN  GOTO 1490
1460  GOTO 1400
1470  REM
1480  REM  SAVE TO SCREEN
1490  TEXT : HOME
1500  PRINT "X, Y, COLOUR, ON/OFF"
1510  PRINT
1520  FOR I = 1 TO P - 1
1530  PRINT C(I,1);",";C(I,2);",";
HC%(I);",";DM%(I)
1540  NEXT I
1550  STOP
1560  REM
1570  REM  SAVE TO TAPE
1580  TEXT : HOME
1590 DM%(101) = P
1600  INPUT "SET TAPE TO RECORD, THEN
PRESS RETURN ";A$
1610  PRINT "COMPUTER WILL GIVE SIX
BEEPS"
1620  STORE C
1630  STORE HC%
1640  STORE DM%
1650  STOP
1660  REM
1670  REM  STORE TO DISC
1680  TEXT : HOME
1690  PRINT  CHR$ (4);"CATALOG"
1700  PRINT : INPUT "FILE NAME ? ";O$
1710  PRINT  CHR$ (4);"OPEN ";O$
1720  PRINT  CHR$ (4);"WRITE ";O$
1730  PRINT P - 1
1740  FOR I = 1 TO P - 1
1750  PRINT C(I,1): PRINT C(I,2): PRINT
HC%(I): PRINT DM%(I)
1760  NEXT I
1770  PRINT  CHR$ (4);"CLOSE ";O$
1780  STOP
1790  REM
1800  REM  READ FIGURE
1810  TEXT : HOME
```

```
1820  PRINT : PRINT "D=DISC T=TAPE
K=KEYBOARD ";
1830  INPUT A$
1840  IF A$ = "D" THEN  GOTO 2040
1850  IF A$ = "T" THEN  GOTO 2230
1860  IF A$ = "K" THEN  GOTO 1900
1870  GOTO 1820
1880  REM
1890  REM  READ FROM SCREEN
1900 PRINT : PRINT "INPUT IN ORDER X, Y,
COLOUR, ON/OFF"
1910  PRINT "LAST ENTRY SHOULD BE
   -1,-1,-1,-1"
1920 P = 1
1930  INPUT "X,Y,C,O ";C(P,1),C(P,2),
HC%(P),DM%(P)
1940  IF C(P,1) < 0 OR C(P,2) < 0 OR
HC%(P) < 0 OR DM%(P) < 0 THEN
GOTO 1980
1950 P = P + 1
1960  IF P = 101 THEN  GOTO 1980
1970  GOTO 1930
1980 O = DM%(P - 1):CL = HC%(P - 1)
1990 P = P + 1
2000  GOSUB 740
2010  GOTO 130
2020  REM
2030  REM  INPUT FROM DISC
2040  PRINT : PRINT  CHR$ (4);"CATALOG"
2050  PRINT : INPUT "WHAT FILE ? ";I$
2060  PRINT  CHR$ (4);"OPEN ";I$
2070  PRINT  CHR$ (4);"READ ";I$
2080  INPUT P
2090 P = P + 1
2100  FOR I = 1 TO P - 1
2110  INPUT C(I,1)
2120  INPUT C(I,2)
2130  INPUT HC%(I)
2140  INPUT DM%(I)
2150  NEXT I
2160  PRINT  CHR$ (4);"CLOSE ";I$
```

```
2170 O = DM%(P - 1):CL = HC%(P - 1)
2180 P = P + 1
2190  GOSUB 740
2200  GOTO 130
2210  REM
2220  REM  INPUT FROM TAPE
2230  PRINT : INPUT "GET TAPE READY,
THEN PRESS RETURN ON COMPUTER, THEN PLAY
ON TAPE RECORDER ";A$
2240  RECALL C
2250  RECALL HC%
2260  RECALL DM%
2270 P = DM%(101)
2280 P = P + 1
2290  GOSUB 740
2300  GOTO 130
2310  REM
2320  REM  START-UP ROUTINE
4000  HGR
4010  HCOLOR= 3
4020  SCALE= 2
4030  ROT= 0
4040  VTAB 24
5000  HIMEM: 38388
5010  POKE 232,245: POKE 233,149
5020  FOR I = 1 TO 10
5030  READ N
5040  POKE 38388 + I,N
5050  NEXT I
5060  DATA  1,0,4,0,40,54,63,36,21,0
5070  RETURN
```

The following arrays are used in this program.

The array C(100,2) holds the co-ordinates of the points — C(,1) is the position across the screen (the X co-ordinate) and C(,2) is the position down the screen (the Y co-ordinate). Note that C(,) is a 'real' array (not an integer

array). Although in *this* program it will only hold integer values, a 'real' array is used so that it is possible to expand this program to include features like magnification and so on, using real numbers to store more accurate positions of the points.

The array HC%(100) holds the color of each point, or rather the color of the line that *ends* at that point.

The array DM%(101) keeps track of which lines are ON and which are OFF. The OFF lines will only result in the position being moved, with no line actually drawn. DM%()=1 indicates that a line is to be drawn, =0 indicates that this line is 'invisible', and should not be drawn. The last element of DM%() (which, you will notice, has 101 elements, rather than 100) is used to store P when the figure is being stored onto tape. P is the next available element of the arrays. So P minus 1 is the number of points. When storing information onto tape, only arrays can be stored, so the variable P is in the last element of DM%() array.

The variables X and Y represent the current cursor position. X is the distance from the left of the screen, Y the distance from the top. CL holds the current color, while O holds the current ON/OFF status (1 for ON, 0 for OFF). CP is the point which was last jumped to by the cursor. When a point is 'moved', it is the one that CP is referring to that will be moved.

The 'start-up' section of the program is in lines 4000 onwards. These lines set the screen into HGR mode, set the color, scale, rotation, and so on, and then POKE the shape of the cursor into the machine's memory.

The program starts off in line 10 by defining the arrays, then goes on to perform the initialization routine (lines 4000 onwards).

Line 30 sets the initial position of the cursor (i.e. the middle of the screen). Line 40 draws it there.

Line 90 sends the program to the routine at line 530, which simply puts the key at the bottom of the screen.

The main loop of this part of the program is from line 130 to line 350. What it does is to take in a particular key entry, interpret and execute it and then go back for another one. The program will continue to go through this loop until the F key is pressed, which will send the program to line 1320.

Line 130 does a GET from the keyboard. This waits for the first key to be pressed, and puts the single character representing that key into A$. The loop then compares A$ to the various relevant keys, and acts accordingly. Lines 140 to 170 deal with the cursor movement keys. They simply add or subtract to X and Y, one screen position at a time. Lines 180 to 210 make the cursor stop at the edge of the screen.

Lines 220 to 290 check for the rest of the important keys. With the exception of F, each has a subroutine that has to be executed when the key is pressed.

Lines 300 to 350 move the cursor when any of the cursor movement keys are pressed. Line 330 erases the old cursor image by XDRAWing it again, and line 340 draws the new one.

The routine at line 370 changes the state of the O variable which shows whether the current line will be ON or OFF. Line 400 sends the program to the subroutine at line 530, which re-prints the key at the bottom of the screen (with the new ON/OFF message).

The routine at line 430 changes the current color which can be from 0 to 7.

The routine at line 600 adds a point. First, line 610 checks to see whether the arrays are full. Although 100 points is quite a lot, the last thing we would want is for the program to crash just when the figure is finished.

Line 620 checks to see whether you are drawing a line of zero length. This can happen if you press the A button twice by mistake, or if you are holding the REPT button down at the time.

Lines 630 to 650 set up the arrays with the correct value, then 660 to 690 draw the new line. Line 700 updates P, and sets the CP pointer to the last point added.

The routine at line 740 erases the last point that has been added by simply reducing the value of P, and re-drawing the figure.

The routine at 910 moves the cursor to the 'previous' point by setting X and Y to the value stored for it in C(,). The routine at 1000 does the same thing, but for the 'next' point.

The routine at line 1080 is the only complicated part of the program. It is the one that 'moves' (or 'fixes') a point. Line 1090 checks that there are some points in the machine to be moved (otherwise this routine will be upset).

Firstly, the routine erases the cursor, getting it out of the way. This is done in line 1110. Then the routine proceeds to erase the lines on either side of the point that is to be moved. What it is going to do is to erase those two lines, and then draw them back in in their new positions. (I could have taken the approach I used in the 'erase' routines, but this is quicker — and since the J key will be used more often than the E key, I've gone to the trouble

to use this quicker method here.) The lines are erased by HPLOTting over them in black. This will leave black dots on any lines that are underneath but that can be rectified later by creating and erasing an extraneous point (thus re-drawing the whole screen).

The purpose of this routine is to get the point in the right place (to complete a curve, for example). It's more important at this stage to re-draw the lines quickly than it is to preserve the ones underneath.

Lines 1130 and 1140 re-draw the previous line in black (HCOLOR=0 is black).

Line 1150 checks to see whether this is the last point to be added (in which case there will be no 'next' line to be moved). If there is no 'next' line then the program jumps to 1170.

1160 HPLOTs the 'next' line in black also.

Lines 1170 to 1190 put the new values into the arrays, including the new color (the current color will be different from the CP point only if you actually use the C key to change it).

Lines 1200 to 1270 re-draw the lines on either side of the point in the correct positions, and then 1280 puts the cursor back onto the screen.

There's nothing remarkable about the rest of the program. When the F key is pressed, it sends the program to line 1320, and you are presented with the choice of either reading in a new figure or storing the current one. Saving and retrieving to and from disc is simple enough. P is put into the file first, then the various arrays. Going to and from the screen/keyboard is fairly simple, too. The only slightly clever bit there is at line 1940, which checks to see whether the 'flag' input has been received. Rather than tell the program how many points you are going to

feed in and then having it count them, it is usually easier to just feed them in one after the other and finish with one that is obviously not right (like −1,−1,−1,−1, as in this example).

Tape storage and recall is via the STORE and RECALL commands, which move arrays on and off tape. Notice that the complete arrays (all 100 points, no matter how few have data in them) are stored and loaded automatically. The top element of DM%() stores the value of P.

Animation with MOVE FIGURE

Having stored two figures, we now want to be able to move smoothly from one to the other. The next program, MOVE FIGURE, makes such animation possible.

The program is easy to use. When RUN, it asks for the source of the first figure (disc, tape or keyboard). Then it inputs that figure, and asks for the source of the next one. When both figures are in memory, the user can specify the number of steps in the animation. The program will generate a series of figures equal to the number of steps plus one.

The only difference between the input section of this program and that of MAKE FIGURE is that when reading a figure from disc the program gives you the option of seeing the CATALOG or not (answer Y or N). This is an improvement over the last method and can be further improved as you'll see later.

Having read in both figures, and been given the number of steps, the program will draw the starting figure

on the screen. It will then wait for a key to be pressed, and put the second one up, and so on.

MOVE FIGURE

```
10  DIM C(100,2),C1(100,2),HC%(100),
DM%(101)
20  PRINT : PRINT "FIRST FIGURE:"
30  GOSUB 360
40 PP = P
50  FOR I = 1 TO P - 1
60 C1(I,1) = C(I,1)
70 C1(I,2) = C(I,2)
80  NEXT I
90  PRINT : PRINT "SECOND FIGURE:"
100  GOSUB 360
110  IF P < > PP THEN  PRINT "FIGURES
HAVE A DIFFERENT NUMBER OF POINTS": STOP
120 PRINT : INPUT "HOW MANY STEPS ? ";S
130  HGR2
140  FOR ST = 0 TO S
150 X = 140:Y = 80
160 P1 = (S - ST) / S:P2 = ST / S
170  FOR I = 1 TO P - 1
180  HCOLOR= HC%(I)
190 OX = X:OY = Y
200  IF I = 1 THEN  HPLOT X,Y
210 X = C1(I,1) * P1 + C(I,1) * P2
220 Y = C1(I,2) * P1 + C(I,2) * P2
230  IF DM%(I) = 0 THEN  HPLOT X,Y:
GOTO 260
240  HPLOT OX,OY
250  HPLOT  TO X,Y
260  NEXT I
270  GET A$
280  HGR2
290  NEXT ST
300  TEXT
```

```
310  PRINT
320  GOTO 120
330  REM
340  STOP
350  REM  READ FIGURE
360  PRINT : PRINT "D=DISC T=TAPE
K=KEYBOARD ";
370  INPUT A$
380  IF A$ = "D" THEN  GOTO 550
390  IF A$ = "T" THEN  GOTO 750
400  IF A$ = "K" THEN  GOTO 440
410  GOTO 360
420  REM
430  REM  READ FROM SCREEN
440  PRINT : PRINT "INPUT IN ORDER X, Y,
COLOUR, ON/OFF"
450  PRINT "LAST ENTRY SHOULD BE
  -1,-1,-1,-1"
460 P = 1
470  INPUT "X,Y,C,O ";C(P,1),C(P,2),
HC%(P),DM%(P)
480  IF C(P,1) < 0 OR C(P,2) < 0 OR
HC%(P) < 0 OR DM%(P) < 0 THEN  GOTO 520
490 P = P + 1
500  IF P = 101 THEN  GOTO 520
510  GOTO 470
520 O = DM%(P - 1):CL = HC%(P - 1)
530  RETURN
540  REM
550  REM  INPUT FROM DISC
560  PRINT : INPUT "CATALOG ? ";A$
570  IF A$ = "N" THEN  GOTO 590
580  PRINT : PRINT  CHR$ (4);"CATALOG"
590  PRINT : INPUT "WHAT FILE ? ";I$
600  PRINT  CHR$ (4);"OPEN ";I$
610  PRINT  CHR$ (4);"READ ";I$
620  INPUT P
630 P = P + 1
640  FOR I = 1 TO P - 1
650  INPUT C(I,1)
660  INPUT C(I,2)
```

```
670  INPUT HC%(I)
680  INPUT DM%(I)
690  NEXT I
700  PRINT  CHR$ (4);"CLOSE ";I$
710 O = DM%(P - 1):CL = HC%(P - 1)
720  RETURN
730  REM
740  REM  INPUT FROM TAPE
750  PRINT : INPUT "GET TAPE READY, THEN
PRESS RETURN ON COMPUTER, THEN PLAY ON
TAPE RECORDER ";A$
760  RECALL C
770  RECALL HC%
780  RECALL DM%
790 P = DM%(101)
800  RETURN
```

The section from line 350 to the end is the same as for the MAKE FIGURE program input section, except for lines 560 to 580, which is the optional CATALOG part. The input section is used as a subroutine to read in the two figures: the first is stored in array C1(,) in lines 50 to 80. If the two figures have different numbers of points, then line 110 will stop the program.

The DM%() and HC%() arrays for the second figure only are used. S is the required number of steps. The variable ST goes from 0 to S, keeping track of what step is being processed.

The main loop of the program is from line 140 to line 290. Each time the loop runs through, P1 and P2 are calculated. P1 and P2 are the proportions of the first and second figures that are to be used. For example, if we have asked for three steps (S = 3), and we are working on the second one (i.e. we are 2/3 of the way through), then P1 will be (3 − 2)/3 = 1/3, and P2 will be 2 / 3 = 2/3. So

we will use 1/3 of the co-ordinates from the first figure, plus 2/3 of the co-ordinates from the second figure. This calculation is done in lines 210 and 220, and will give points which move in equal steps from their positions in the first figure to their positions in the second figure.

Lines 230 to 250 actually draw the figure on the screen. Line 270 accepts a key press, thus making the program wait until you have had a good look at the figure.

The practical use of this program will probably depend on the availability of a printer. The printed version (or perhaps one traced off the screen) can be used as the basis for a drawn final image, or it can be used 'raw'. Of course, you don't have to go to the expense of filming these to make animation; the old 'flick' method works just as well. You can make up a small book with one image per page, and then by bending back the pages and letting them go one at a time you can make them 'flick' flat, revealing each image in turn.

Figure 2–21 is an example of the use of the program.

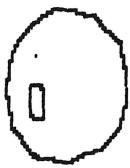
In the first image, the ball is at the start of a fall. A reflection on the side is about two-thirds of the way down. The ball falls, and by the time we reach image 4, it has hit the surface. The reflection has moved up the surface by this time, showing us the position of the light source. In the final images, the ball 'squashes', and ends up fairly flat in image 7. The reflection is also flattened.

This sequence (which could easily have been turned into 100 images for a slow fall) was generated by first making image 4 and saving it. By altering image 4, image 1 was obtained. Each point on the ball in turn was moved using only the U key thus ensuring that each point

(1)

(2)

(3)



(4)

(5)

(6)

(7)

Figure 2–21

would stay in the correct position during the fall. Starting from image 4 again (having saved image 1), image 7 was obtained by alterations using only the M key to move the points.

In each of the three images, the reflection was also altered so that it was in the correct position.

The sequence of seven images was generated by the MOVE FIGURE program in two steps: firstly to go from 1 to 4, and then again to go from 4 to 7. The sequence of images could have been extended indefinitely, taking the last image and altering it to give the next one, and then producing a smooth set of images between them.

Three Dimensions

Chapter 3

In this section we develop programs to generate animation in three dimensions.

First of all we need to develop a routine that will put a 'view' of a solid figure onto the screen. The three-dimensional figure will be stored in memory in the same way as the 'flat' figures, as a series of numbers.

In the 'flat' figures, each point was represented by the two co-ordinates X and Y being the distance from the left and top of the screen respectively.

To specify a point in three dimensions we need a third co-ordinate, Z, which will indicate how far the point is from the *back* of the screen.

Let us imagine a point in space which is at the top left hand corner of the screen, and right at the 'back'. If you imagine the screen to be the front of a square glass tank, the point would be at the top left of the part of the tank furthest away from you. This point will be represented by the co-ordinates 0, 0, 0. It is at a distance of zero from the left and the top of the screen and also a zero distance from the back (X=0,Y=0,Z=0).

Some sample points will make this clear. The point 80, 0, 0 would be half way across the back at the top. The point 80, 80, 0 would be in the middle of the back. The point 80, 80, 80 would be somewhere in the middle of the tank.

Imagine a solid figure somewhere in the tank. It can be described by a series of these sets of three 'co-ordinates'. Looking through the front of the tank, the screen, the solid figure would look different according to our 'point of view'. By changing this imaginary point of view, we would see different parts of the figure viewing it from different angles.

In order to draw a flat image of this solid figure, we 'project' it onto the screen: we represent each of the three dimensional points by an appropriate two dimensional point. To do this we need to know the co-ordinates of each of the points of the three dimensional figure, the co-ordinates of our 'point of view' and the direction in which we are looking from our point of view.

Over the next few pages a routine will be developed that will put onto the screen the view that we will see, given the information just mentioned. First, how do we define the direction in which we are looking? It's really quite easy. We define direction on a flat page in terms of an angle. The direction that points from left to right is defined as zero. An angle that is anti-clockwise from that is greater than zero, and one clockwise from it is less than zero. The computer measures angles in radians rather than degrees (an angle of 3.14 radians equals 180 degrees). So if 'left to right' is zero, then 'up' will be about

1.57 radians, 'down' will be −1.57 and 'right to left' will be −3.14.

To define direction in three dimensions, imagine there is a little arrow in the tank pointing in the direction we're interested in. If we look from directly in front of the screen, we see the arrow as pointing in a particular direction, while if we look from directly above the tank, we see the arrow pointing in a different direction. We can describe the direction of the arrow using two numbers: one to describe the direction as it looks from the front; and the other to describe the direction as it looks from above. The two numbers that will define the direction will be angles and we will call the 'from on top' angle theta, and the 'from the front' angle phi (TS and PS for short, in the program).

Any given view of the three dimensional object or figure can now be defined in terms of the co-ordinates of the point of view, and in terms of the two angles, theta and phi, of the direction of view.

We also need to define the 'magnification' of our view in order to be able to 'blow up' sections of the figure.

Having defined the figure by its co-ordinates, and also the point and direction of viewing, we need a way of plotting this image on a screen. Imagine a 'screen' between the point of view and the figure, the point of view being an eyeball looking in the direction of view. To get an image of the figure on the screen, we would draw a line from each point on the figure to the eyeball and plot all the points where the lines hit the screen. Each of the points in the figure will fall at a particular point on the screen.

In principle, this procedure is like looking at some view through a window and, without moving, tracing the image of the view on the window pane.

We want to draw images on the screen of the TV or monitor so that when we plot the 'apparent' position of each of the points of the figure onto the screen, it will look as if the figure is actually behind the screen of the TV.

Vectors

In order to solve the problem we have posed, we need to use vectors. Unfortunately, we can't spend time here exploring this fascinating subject. If you want to learn more about it, you'll have to get a maths book. All we can do here is to give an empirical (that is, 'rule of thumb') method for using vectors to solve this problem.

A vector is something used by engineers to talk about a difference in position between two points. Each vector is made up of three 'components' — an X, a Y and a Z component. The X component is the difference in left-right position, the Y component in up-down position, and the Z component in back-front position. The vector that gives the difference in position between two points is said to 'join' the two points.

If we have a point near the top back left of the tank, its co-ordinates might be 10, 15, 5. A point near the middle might have co-ordinates 60, 80, 70. Calling the

first point A and the second B, the vector, AB, that joins them would have the components 50, 65, 65 given by

```
X=60-10=50
Y=80-15=65
Z=70-5=65.
```

So the components of a vector are the changes in position in the three directions, X, Y, and Z that it represents from its beginning to its end.

It is useful for our purposes to find the components of a unit vector. A unit vector is a vector that has a length of 1, and in order to find its components we need to know the length of the large vector of which it is a part.

The length of a vector is found from its components by using the theorem of Pythagoras.

```
length=SQR(X*X+Y*Y+Z*Z)
```

So the length of our vector AB is:

```
length of AB=SQR(50*50+65*65+65*65)
             =104.64
```

The 'unit vector' of AB is a vector which is in the same *direction* as AB, but which has a length of 1. If it is in the same direction, then its components must be in the same proportion to one another as the components of AB are.

We find the unit vector of AB by dividing each of the components of the vector AB by the length of AB, giving:

```
X=50/104.64=0.479
Y=65/104.64=0.621
Z=65/104.64=0.621
```

Let's just check that the length of this unit vector is 1.

```
length=SQR(0.479*0.479+0.621*0.621+0.621*0.621)
       =1.000
```

Suppose now that we have two unit vectors in different directions, and we want to find a third unit vector which is at right angles to *both* of them? Let's call one unit vector vector 1, the other unit vector vector 2, and the vector that we're trying to find (at right angles to them both) vector 3, then the equations for finding 3 from 1 and 2 are:

```
X3=Y1*Z2-Z1*Y2
Y3=Z1*X2-X1*Z2
Z3=X1*Y2-Y1*X2
```

They give us a vector (although not a unit vector) in the direction of vector 3. Then the unit vector 3 is given by:

```
D=SQR(X3*X3+Y3*Y3+Z3*Z3)
X3=X3 / D
Y3=Y3 / D
Z3=Z3 / D
```

The last four lines are not needed if vector 1 was at right angles to vector 2 in the first place (but this is not necessarily the case).

The angle between two unit vectors is given by:

```
CA=X1*X2+Y1*Y2+Z1*Z2
```

CA is the *cos* of the angle we're looking for. To find the actual angle itself, you find the inverse cos of CA — but this is seldom necessary, as the cos of the angle is often all we need.

If we have two vectors AB and BC (the vector from B to a third point, C), adding the components of AB to the components of BC will give us the vector AC. So if we add the X component of AB to the X component of BC, we get the X component of AC, and so on.

By multiplying the unit vector of AB by the *length* of AB, we obtain AB. This is not surprising, as we found the unit vector originally by dividing AB by its own length. But it is worth bearing in mind (especially in the case of vector 3 at the top of the page, which we did not find from its larger 'parent' vector).

Projecting vectors

We will also need to know how to find the 'projection' of a vector in the direction of another vector. If we started both vectors from the same point, and then looked at right angles to one of the vectors, how long would the other vector appear in the direction of the first? To start with, we have to find the unit vector of one of the original vectors. Now we have one unit vector and one full vector. What we are trying to find is how far in the direction of the unit vector, the full vector stretches.

We proceed as if we were trying to find the angle between two unit vectors (except that in this case, one of the vectors is not a unit vector):

$$D = X1 * X2 + Y1 * Y2 + Z1 * Z2$$

If you think about it, you will see that the answer we were trying to find was really the cos of the angle between the vectors, times the length of the full vector. So all we have done is to substitute D, the projected distance, for CA, the cos of the angle, and also to substitute a full vector for its unit vector in the other side of the equation. We can now use vectors to solve our problem.

PROG 3

```
1140  REM  ROUTINE TO PUT VIEW ONTO
SCREEN
1150 X8 =  COS (TS) *  COS (PS)
```

```
1160 Y8 =  SIN (PS) *  COS (TS)
1170 Z8 =  SIN (TS) *  COS (PS)
1180 D1 =  SQR (X8 * X8 + Y8 * Y8 + Z8 * Z8)
1190 X8 = X8 / D1:Y8 = Y8 / D1:Z8 = Z8 / D1
1200 E1 = X8 * R:F1 = Y8 * R:G1 = Z8 * R
1210  FOR I = 1 TO NP
1220 O%(I) = 0
1230 X3 = A(I,1) - XE:Y3 = A(I,2) - YE:
Z3 = A(I,3) - ZE
1240 D =  SQR (X3 * X3 + Y3 * Y3 + Z3 * Z3)
1250 X3 = X3 / D:Y3 = Y3 / D:Z3 = Z3 / D
1260 CA = X8 * X3 + Y8 * Y3 + Z8 * Z3
1270  IF CA < 0 THEN O%(I) = 1
1280 D2 = R / CA
1290 X4 =  - E1 + D2 * X3:Y4 =  - F1 +
D2 * Y3:Z4 =  - G1 + D2 * Z3
1300 X =  COS (TS + PI / 2) * X4 +
SIN (TS + PI / 2) * Z4
1310 SS =  SGN (Y4): IF SS = 0 THEN SS =
-  SGN (X4)
1320 Y =  SQR (X4 * X4 + Y4 * Y4 + Z4 * Z4
- X * X + 1E - 5) * SS
1330  IF X > 32000 THEN X = 32000
1340  IF X <  - 32000 THEN X =  - 32000
1350  IF Y > 32000 THEN Y = 32000
1360  IF Y <  - 32000 THEN Y =  - 32000
1370 B%(I,1) = 140 + X:B%(I,2) = 80 + Y
1380  NEXT I
1390  HGR2
1400 F = 0
1410 X = B%(1,1):Y = B%(1,2)
1420  GOSUB 6000
1430  IF NP = 1 THEN  RETURN
1440  FOR I = 2 TO NP
1450 F = 1
1460  IF O%(I) = 1 OR O%(I - 1) = 1 OR
C%(I) < 0 THEN F = 0
1470  IF F = 1 THEN  HCOLOR= C%(I)
1480 X = B%(I,1):Y = B%(I,2)
1490  GOSUB 6000
1500  NEXT I
```

The variables TS and PS are the angles representing the 'from the front' and 'from the top' directions of our direction of view.

X8, Y8 and Z8 are the components of the unit vector in our direction of view.

R is the distance from our point of view to the 'screen'. This gives the magnification. As the screen gets further from our point of view, the image on it gets bigger.

E1, F1 and G1 are the components of a vector from our point of view, in the direction of view, to the screen. If you like, they are the components of a vector from the point of view to the point nearest it on the screen — the middle of the screen.

NP is the number of points in the figure.

A(,) is the array that holds the co-ordinates of each of the points in the figure. A(,1) holds the X-co-ordinates, A(,2) the Y and A(,3) the Z-co-ordinates. So A(1,1) holds the X-co-ordinate of point 1, and there are NP points.

XE, YE and ZE are the co-ordinates of our point of view (the 'eye').

X3, Y3 and Z3 hold the co-ordinates of a unit vector from our point of view to a given point on the figure.

CA is the cos of the apparent angle from the middle of the screen to the point we're considering, from the point of view. If you like, it's the angle between vector 8 (which is X8, Y8, Z8) and vector 3 (X3, Y3, Z3).

D2 is the distance from the point of view to the place on the screen where the point of the figure appears to be.

X4, Y4 and Z4 hold the components of the vector from the middle of the screen to the place on the screen where the point of the figure appears to be.

B%(,) holds the actual TV screen co-ordinates of the point we're after. B%(,1) holds the X-co-ordinate, B%(,2) the Y-co-ordinate. HPLOTting the elements of B%(,) successively on the screen will give us our final view of the figure on the monitor screen.

F is the flag that we use to tell the plotting routine (called EXPLOT, which we developed earlier in the book) whether this is a plot from the previous point, or just a move.

How the program works

Lines 1150 to 1190 convert the direction of view into vector 8, a unit vector that points in the same direction. Why not just hold the direction in terms of the unit vector in the first place? Because it is easier to control two separate angles (as you'll see when you come to operate the full program) than to tell the computer which direction to look in all in one go. 1150 to 1190 convert PS and TS into vector 8 by first finding the vector that points in the direction of PS and TS (in lines 1150 to 1170) and then converting it into a unit vector.

Now imagine a line which has a length of one. We start with it pointing from left to right. The length of it as seen from the front of the screen will be 1. If we rotate it *into* the screen, it will seem shorter and shorter as it goes towards the back. In fact, the apparent length of it will be equal to the cos of the angle by which it is rotated. Similarly, if we look at the line from the top, and rotate it downwards, then the apparent length will be equal to the cos of the angle by which we have rotated it downwards.

So the 'shortening' of the line will be proportional to the cos of the angle through which it is rotated.

In order to find the X-component of vector 8, we apply two factors: the cos of TS and the cos of PS, in line 1150. When PS and TS are both 0, the line is pointing from left to right and the X-component is 1. As either angle increases, the apparent length of the line will shorten. The X-component of the vector is actually the distance in the X direction from one end of it to the other, so the X-component will be proportional to the cos of TS and the cos of PS. The other components are found in a similar manner.

So now we have vector 8, the unit vector in the direction of view. The middle of the screen lies in the direction of view from the point of view, and is a distance R away. So it is a simple matter to find E1, F1, G1, which is the vector from the point of view to the middle of the screen. This is done in line 1200.

Now we come to the main loop of the routine — from line 1210 to line 1380. Inside this loop, we are looking at particular points of the figure.

Line 1220 sets O%(I) to 0. O%() will hold information about which points are out of the field of view and which are in.

Line 1230 finds the components of vector 3, from the point of view to the point on the figure, by finding the difference between the co-ordinates of one and the co-ordinates of the other.

Line 1240 finds the distance between the point of view and the point on the figure, and line 1250 uses this to convert vector 3 into a unit vector pointing from the point of view to the point on the figure.

Line 1260 calculates the cos of the angle between vector 3 and vector 8. This gives us the angle that the point on the figure is from the middle of the screen, as seen from the point of view.

Line 1270 checks to see whether the point is actually *behind* the point of view. This would mean that it is not visible. Line 1270 makes sure that these invisible points are not drawn by checking the angle.

If CA (the cos of the angle that the point on the figure is from the middle of the screen) is *less* than zero, then the angle is more than 90 degrees. We can say that because of the curve that cos makes, it is above zero from plus 90 degrees to minus 90 degrees.

If a point is not visible, then the corresponding element of O%() is set to 1. This will be of use to us later.

Line 1280 finds D2, the distance from the point of view to the point where the figure point appears on the screen. We already know R, which is the distance from the point of view to the screen. And we have just found CA, which is the cos of the angle between the middle of the screen, the point of view, and the point on the figure. By simple geometry we can find out how far it is from the point of view, along a line towards the point on the figure, to the screen. This distance we will multiply by the unit vector from the point of view to the point on the figure (vector 3) giving us the vector (no longer a unit vector) from the point of view to the point we want on the screen.

Now we already have the vector from the point of view to the middle of the screen (it is E1, F1, G1). So after we have worked out this multiplication, we can by

subtracting E1, F1, G1 from the result, find the vector from the *middle* of the screen to the *point* on the screen that we want. This is all done in line 1290.

Line 1290 multiplies vector 3 by D2, and then subtracts E1, F1, G1, to give vector 4. Vector 4 is now the vector from the middle of the screen to the point on the screen that we want. All we have to do now is to 'resolve' this vector, that is to find out how far up, and how far across the screen we have to go to plot the point.

Now we need to find X and Y, which tell us where on the TV screen the point will have to be HPLOTted. To do this, we take vector 4, and find its projection onto the unit vector that points from left to right across the screen.

Looked at from the top, the direction of view has an angle of TS. Also from the top, the unit vector that points across the screen will have an angle of TS plus 90 degrees. So the X and Z components of this unit vector will be given by:

```
X=COS(TS+PI / 2)
Z=SIN(TS+PI / 2)
```

Since we earlier set PI to 3.14, PI/2 is equivalent to 90 degrees. The Y component will be zero since the unit vector we're looking at is horizontal.

To find the length of projection of the vector from the middle of the screen to the point on the screen we're after, in the direction of this unit vector, we do the

calculation in line 1300. This is just the calculation I showed you earlier to find the projection of a vector in the direction of a unit vector — except that the Y part has been left out, due to the fact that the result of the Y part will always be zero.

Having found variable X, which is in fact the distance from the centre of the TV screen across to the point to be plotted, we find Y, which is the direction *down* the screen from the middle to the same point. This is done using the theorem of Pythagoras. In line 1320, we find the length of one of the short sides of the triangle from the length of the hypotenuse and the length of the third side. The 'fiddle factor' of .00001 is to prevent embarrassment due to the inaccuracy of the arithmetic when Y is very nearly zero.

Line 1310 gives us the direction of Y. You will notice that without the SS variable, line 1320 would always give an answer greater than zero!

So by the time we reach line 1330, we have both X and Y, which give us the position on the TV screen for the point of the figure that we're working on this time through the loop.

Lines 1330 to 1360 prevent an overflow in line 1370, since integer variables can only hold values up to about plus or minus 32000.

Line 1370 puts the results into B%(,), after first adding the X and Y co-ordinates of the centre of the screen. B%(,) is now ready for HPLOTting.

Lines 1410 and 1420 put the first point onto the screen. The first point is a special case, since it will always be plotted with a value of F which is 0. This is so that the computer will not attempt to draw a line from a non-existent 'previous' point.

Line 1460 checks to see whether the whole of the line we are about to draw is in view. If the line passes out of view (i.e. *behind* the point of view), then it will not be drawn.

Line 1460 checks not only that the point we're about to plot is in view, but also that the point *before* it is in view. This is necessary to prevent the situation occurring where the computer tries to draw a line to a point which is in view from a point which is *out* of view. Line 1460 also checks the color of the line — I've used the color array, C%(), to keep track of which lines are 'invisible' by marking them with a 'color' less than zero. So if C%() is less than zero, the line is invisible. This 'invisible' line system is necessary for the same reason that it was used in the two-dimensional program earlier.

The routine calls the EXPLOT routine which starts at line 6000. This is the same as the one we developed earlier, which allows us to draw lines which start and finish over the edge of the screen border (although not behind our point of view).

Now for the program proper, with a few words first on how to operate it.

Using THREE-D FIGURE

When you RUN the program, it will first ask you whether you want to input an old shape, or start a new one. The first time through, of course, you will want a new shape. Press N, then RETURN. The screen will clear and a cursor (diamond-shaped) will appear at the middle. Now press

V. After a short delay, the screen will clear, and you will see Figure 3-1.

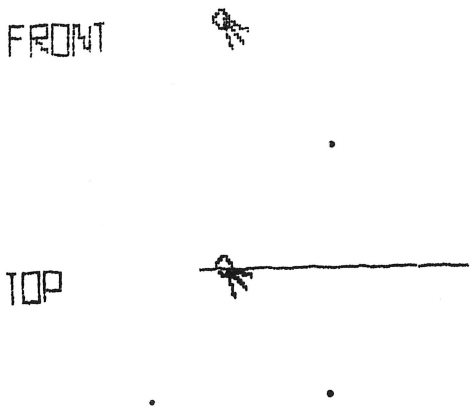


Figure 3-1

This is a 'map' showing the size and shape of the figure, where the point of view is and what the direction of view is. Press V again. You will now see the cursor in the middle of the screen again. The V key causes the screen to go from the 'map' to the view of the figure and back again.

Now press the C key. A message will appear at the top of the screen, telling you what the current color is (it will be 0, which is black). Enter a color of 3 (white) and press RETURN. Again, you will see the cursor.

Now move the cursor around the screen using the U, H, M and K keys and the REPT key in the same way as with the two-dimensional program we developed earlier. Put the cursor somewhere in the top right hand corner of the screen and press the A key. This will add a point to the figure. A white line should appear, from the centre of the screen to the cursor.

Move the cursor again, this time to the left of the screen. Press the A key again. Another line should appear, from the end of the previous line to the new cursor position. Now move the cursor down to near the bottom of the screen, and add another line by pressing A again.

Now press the P and Q keys to see what effect they have. They move the cursor from one point to another, in the order in which they were entered.

Using the P (for 'previous') key, move the cursor to the first point (which is in the middle of the screen). Now press the Z key. This will move the cursor to the last point that was entered. This is useful because when a point is added, the resulting line is from the last point entered to the cursor position. So it is sometimes necessary to identify the last point entered quickly, and the Z key allows you to do this. Now press the E key. The screen will blank, and the figure will be re-drawn with the last point erased.

Add another couple of points to the figure. Then using the H, M, K, U and REPT keys, move the cursor so that it is over the point near the top right of the screen. Press the J key. If you hear three beeps, then the cursor is not quite over the point. Move it again carefully, and press the J key. Keep trying until no beeps sound.

What you have done is to 'mark' one of the points. This can also be done by using the P and Q keys to go through each point in turn until you spot the right one, but this method (using the J key) is sometimes necessary.

Now move the cursor down slightly, and press the F key. This will move the point that you have just 'marked' to the place where the cursor is. New lines will be drawn

to the previous and next points from the one that was moved. The old lines will not, however, be erased — this is to speed up the operation of the program. In this mode, it is more important to be able to see the results of moving the point than it is to see the figure 'clean'.

Press the space bar (that's the bar at the bottom of the keyboard). The screen will clear, and the figure will be re-drawn with the old lines erased. Press the space bar again. Again, the screen will clear and the figure be re-drawn. Pressing the space bar will always clear the screen in this way.

Now press the V key again. You will see something like Figure 3–2.

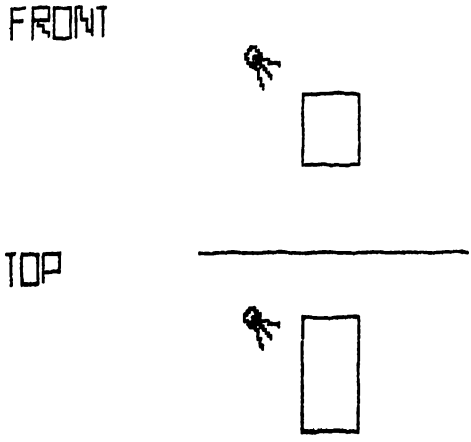


Figure 3–2

The two squares at the top and bottom are the figure. Notice the line that goes across the screen from left to

right. Above this line is a view from the front — the figure lies inside the square, and the point of view is up at the left. The point of view is the small diamond-shape with the three radiating lines.

Below the line is a view from the top. Again, the figure is inside the box, and the point of view is at the left behind the figure.

The lines coming out of the point of view in the FRONT and TOP parts of the screen show the direction of view. Notice that there are *three* lines coming out of the point of view. The middle line shows the actual direction of view, and the two to either side of it show the limits of the screen (roughly). So by changing the magnification, we should change the position of these two lines.

Press the B (for 'bigger') key once. The two 'screen limit' lines on the point of view should move apart slightly. Again, the old lines will stay there. Press the space bar. The screen will be re-drawn with the old lines removed. Again, the space bar gives a 'clean' view.

Now press the V key. The figure that you drew should appear, only slightly smaller. This is as expected — you changed the magnification.

The B key makes the field of view bigger (i.e. reduces the magnification). The S key (for 'smaller') makes the field of view smaller by *increasing* the magnification. Now press V again to get back to the 'map'. Press the H key (yes, it has a different function in the 'map' mode). The point of view should move to the left. The radiating lines will not move, however.

Press the H key again several times. After about the sixth press, the screen will clear and the 'map' will be re-drawn.

There is a counter in the program that clears the screen every so often, so that you do not get confused with old images of the point of view.

Press the O key. The point of view will move back to where it was before you started moving it. The O key (for 'old') brings back the last 'saved' point of view position and magnification. I'll show you how to 'save' that old point of view setting later.

Now try using the I, U, M and N keys. They move the FRONT and TOP points of view up and down the screen. If you think about it, you will see why we need the *three* pairs of keys (H&K, I&N, U&M) to move the point of view around.

Press the R key. The direction of view of the FRONT point of view should move clockwise (again, the old lines will remain until you clear them, or until the machine does). The R key moves the FRONT direction of view clockwise, and the E key moves it the other way. The F and D keys do the same for the TOP direction of view.

Press the O key again to get the old direction of view back. Now press the V key. Notice where the figure falls on the screen. Press the V key, and then press the R key to move the FRONT direction of view clockwise. Before pressing the V key again, take a look at the FRONT part of the screen. Think about where the image of the figure will be now. Since we have moved the direction of view downwards (remember that this is a view from the front of the figure), the image of the figure should have moved *upwards*. Press the V key.

Experiment with the various keys for moving the point of view and changing the direction of view.

Remember that all of the keys have different functions in the two modes.

If you 'lose sight' of the figure, you can always bring it back by making sure that both of the directions of view (TOP and FRONT) point towards the boxes that show the limits of the figure. That is what the 'map' is for.

When in the 'view' mode (as opposed to the 'map' mode), press the S key (for 'save'). You will get this message:

```
T=SAVE TO TAPE
S=SAVE TO SCREEN
D=SAVE TO DISC

C=CANCEL SAVE
Q=QUIT (LOSE CURRENT FIGURE)
```

Press the C key, then RETURN. After a short delay, you will get back to where you were with no change. Press S again, and this time press S in response to the above message. On the screen will appear a list of the co-ordinates of each of the points, the line colors, the co-ordinates of the point of view, and so on. Pressing RETURN will get you back to where you were.

If you had pressed the T or D keys instead of the S (SCREEN) key, you could have saved the whole figure to tape or disc, along with the current position and direction of view, and the current magnification.

Press S to get the message, then press Q to terminate the program. You will lose the figure that you have been working on, and the screen will show the message:

```
BREAK IN 8120
```

RUN the program again and enter N to start a new figure. Press V, and use the appropriate commands to move the point of view to the position shown in Figure 3–3.

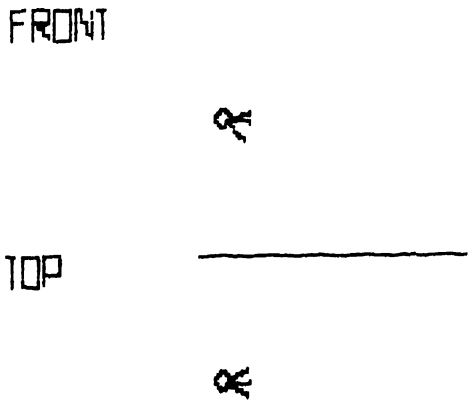


Figure 3–3

Now go back to the view, and draw a triangle, like the one shown in Figure 3–4 (remember to use the C key to change the color to 3).

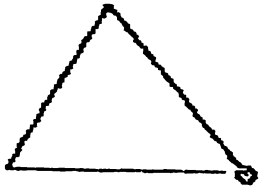


Figure 3–4

Back to the map again, and you will see Figure 3–5.

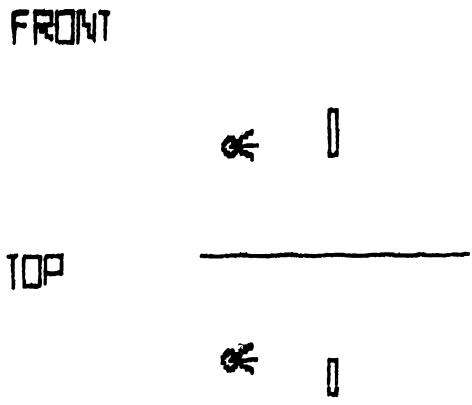


Figure 3–5

The vertical lines show the shape of the figure (which is a triangle set vertically, with no depth from left to right). Move the direction of view to here:

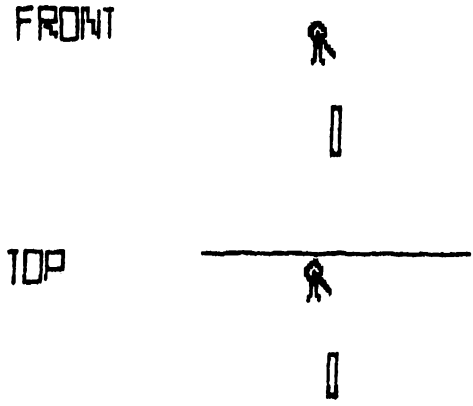


Figure 3–6

Now press V, and you should see something like this — this is the triangle seen from the top:

Figure 3–7

Now using the Z key, move the cursor to the bottom point of the figure (which is the last one you added).

Using the H, U, M and K keys, move the cursor so that it makes the third point of a triangle with the outer two points of the existing figure. Press the A key to add a line.

Now go back to the 'map', and move the cursor to get back to the position that you drew the first triangle of points from. Press V again, and you will get this:

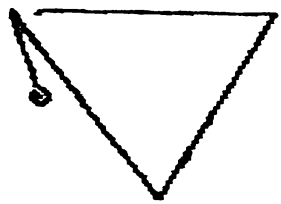


Figure 3–8

When you were looking from the top of the figure, and added a point, *it was added at the same distance from the point of view as the last point 'marked'* (which, in this case, also happened to be the last point added).

Clear the memory (by quitting the program using the S key, and then RUNning it again). Change the color to 3, then put a line of points vertically on the screen at intervals of about a centimetre (half an inch).

Go to the 'map' by pressing V, then move the point of view so that you are looking at the figure from a different direction, then press V again. What you have done is to make a curve, whose centre is the point of view (at the position it was in when you added the points). This facility is useful — it means that you can construct curves using line segments. Unfortunately, it is limited by the fact that you cannot position the point of view very accurately.

Now re-RUN the program, and respond to it in the following way (which will result in your entering a figure into the machine via the keyboard):

```
I=INPUT AN OLD FIGURE
N=START A NEW FIGURE

?I


T=INPUT FROM TAPE
K=INPUT FROM KEYBOARD
D=INPUT FROM DISC

?K

INPUT IN ORDER X, Y, Z, COLOUR
LAST ENTRY SHOULD BE 9,9,9,9
```

```
X,Y,Z,C 0,-5,-10,-1
X,Y,Z,C 0,-5,3,3
X,Y,Z,C 0,5,3,3
X,Y,Z,C 0,5,-10,3
X,Y,Z,C 0,-5,-10,3
X,Y,Z,C 5,-5,-10,3
X,Y,Z,C 5,-5,3,3
X,Y,Z,C 0,-5,3,3
X,Y,Z,C 0,5,-10,-1
X,Y,Z,C 5,5,-10,3
X,Y,Z,C 5,5,3,3
X,Y,Z,C 0,5,3,3
X,Y,Z,C 9,9,9,9
X,Y,Z OF VIEW ? -30,-30,-20
HORIZ AND VERT ANGLE ? .6,.8
MAGNIFICATION ? 300
```

The machine should respond with the view shown in Figure 3-9.

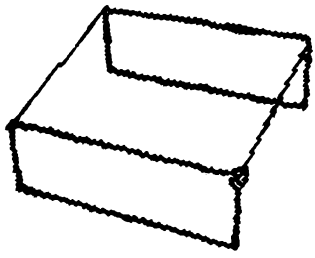


Figure 3-9

Now press the V key, and you should get the map shown in Figure 3-10.

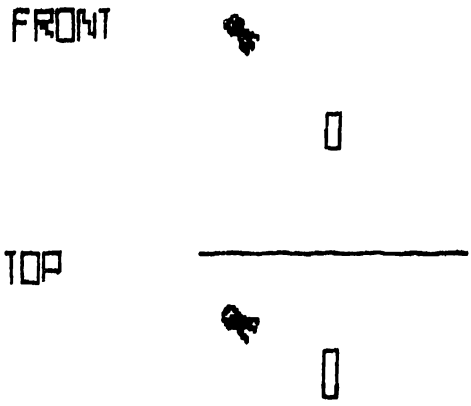


Figure 3–10

Press the X key. The map will change to that shown in Figure 3–11.

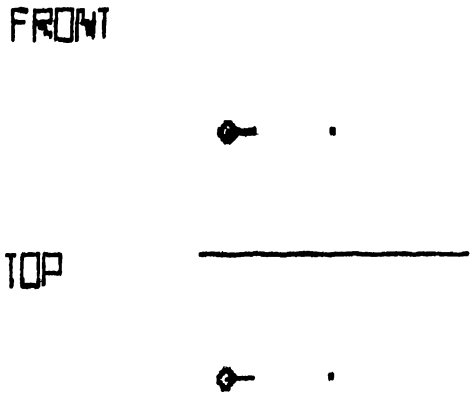


Figure 3–11

What we are doing here is looking at the figure along the X-direction, and from far enough away that perspective has little effect on the view. This sort of projection is often used in technical drawing (drafting).

Press the O key. The point of view will move back to where it was. Now press the Y key. The point of view will change so that we are looking at the figure from above. Now press the O key again (this is important, as you will see later), and press the Z key. This gives us a view of the figure from the front.

Press the O key to get the old point of view back. Now use the E key to rotate the direction of view around to face the other way (it doesn't matter very much which way). Press the X key, and then press the O key. Notice that we have changed the 'old' stored value of the point of view and the direction of view. This 'old' value is stored when the program is first started, and when an X, Y or Z key is pressed.

THREE-D FIGURE

```
1  DIM P(2),Q(2):X9 = 279:Y9 = 191
2 P1 = 0.1:N1 = 1:N2 = 2
10 PI = 3.14159265:CC% = 3
20  GOSUB 5000
30  HIMEM: 16384
40  DIM A(150,3),B%(150,2),O%(151),
C%(150),PP(7),OP(7):MP = 150
50  TEXT : HOME
60  PRINT : PRINT : PRINT
70  PRINT "I = INPUT AN OLD FIGURE
80  PRINT "N = START A NEW FIGURE
90  PRINT : INPUT A$
```

```
100 NP = 1:XE =  - 100:YE =  - 100:ZE =
  - 100:PS = PI / 4:TS = PI / 4:R = 200
110 A(1,1) = 0:A(1,2) = 0:A(1,3) = 0:C%(1) = 0
120 J = 1
130  IF A$ = "N" THEN  GOTO 170
140  GOSUB 7000
150 NP =  INT (PP(1)):XE = PP(2):YE =
PP(3):ZE = PP(4)
160 TS = PP(5):PS = PP(6):R = PP(7)
170 OP(2) = XE:OP(3) = YE:OP(4) =ZE:
OP(5) = TS:OP(6) = PS:OP(7) = R
180  GOSUB 1140
190  IF J <  > 0 THEN X = B%(J,1):Y =
B%(J,2):CC% = C%(J)
200  IF X < 1 OR X > X9 THEN X = 140
210  IF Y < 1 OR Y > Y9 THEN Y = 80
220  SCALE= 1
230  VTAB (24)
240  XDRAW 3 AT X,Y
250  GET A$
260  XDRAW 3
270  IF A$ = "U" THEN Y = Y - 1: IF Y < 1
THEN Y = 1
280  IF A$ = "M" THEN Y = Y + 1: IF Y > Y9
THEN Y = Y9
290  IF A$ = "H" THEN X = X - 1: IF X < 1
THEN X = 1
300  IF A$ = "K" THEN X = X + 1: IF X > X9
THEN X = X9
310  IF A$ = "U" OR A$ = "M" OR A$ = "H"
OR A$ = "K" THEN  GOTO 240
320  IF A$ = "V" THEN  GOTO 450
330  IF A$ = "F" THEN  GOTO 590
340  IF A$ = "A" THEN  GOTO 810
350  IF A$ = "S" THEN  GOTO 8000
360  IF A$ = "J" THEN  GOTO 480
370  IF A$ = "C" THEN  GOTO 1020
380  IF A$ = "Z" THEN J = NP
390  IF A$ = "Q" THEN J = J + 1:
IF J > NP THEN J = NP
```

```
400  IF A$ = "P" THEN J = J - 1:
IF J < 1 THEN J = 1
410  IF A$ = "E" THEN NP = NP - 1:
O%(NP + 1) = 0: IF NP < 1 THEN NP = 1
420  IF A$ = "E" OR A$ = " " THEN
GOTO 180
430  IF A$ = "P" OR A$ = "Q" OR A$ = "Z"
THEN  GOTO 190
440  GOTO 240
450  GOSUB 1530
460  GOTO 180
470  REM
480  REM  FIND MARKED POINT
490 J = 0
500  FOR I = 1 TO NP
510  IF B%(I,1) = X AND B%(I,2) = Y
THEN J = I:CC% = C%(I): GOTO 240
520  NEXT I
530  FOR II = 1 TO 3
540  PRINT  CHR$ (7): REM  BELL
550  FOR I = 1 TO 100: NEXT I
560  NEXT II
570  GOTO 240
580  REM
590  REM  MOVE POINT
600  IF J = 0 THEN  GOTO 530
610 XX = X - 140:YY = Y - 80
620 D =  SQR ((A(J,1) - XE) ^ 2 +
(A(J,2) - YE) ^ 2 + (A(J,3) - ZE) ^ 2)
630 X6 =  COS (TS + PI / 2):Z6 =
SIN (TS + PI / 2)
640 X7 =  - Z6 * Y8
650 Y7 = Z6 * X8 - X6 * Z8
660 Z7 = X6 * Y8
670 X5 = E1 + X6 * XX + X7 * YY:Y5 =
F1 + Y7 * YY:Z5 = G1 + Z6 * XX + Z7 * YY
680 D = D /  SQR (X5 * X5 + Y5 * Y5 +
Z5 * Z5)
690 A(J,1) = XE + X5 * D
700 A(J,2) = YE + Y5 * D
710 A(J,3) = ZE + Z5 * D
```

```
720 C%(J) = CC%:O%(J) = 0
730 B%(J,1) = X:B%(J,2) = Y:F = 1
740  IF C%(J) < 0 OR O%(J - 1) = 1 THEN
GOTO 760
750  HCOLOR= C%(J):OX = X:OY = Y:X =
B%(J - 1,1):Y = B%(J - 1,2): GOSUB 6000
755 X = B%(J,1):Y = B%(J,2)
760  IF J = NP THEN  GOTO 790
770  IF C%(J + 1) < 0 OR O%(J + 1) = 1
THEN  GOTO 790
780 HCOLOR= C%(J + 1):OX = B%(J + 1,1):
OY = B%(J + 1,2): GOSUB 6000
785 X = B%(J,1):Y = B%(J,2)
790  GOTO 240
800  REM
810  REM  ADD A POINT
820  IF J = 0 THEN  GOTO 530
830 XX = X - 140:YY = Y - 80
840 D =  SQR ((A(J,1) - XE) ^ 2 +
(A(J,2) - YE) ^ 2 + (A(J,3) - ZE) ^ 2)
850 X6 =  COS (TS + PI / 2):Z6 =
SIN (TS + PI / 2)
860 X7 =  - Z6 * Y8
870 Y7 = Z6 * X8 - X6 * Z8
880 Z7 = X6 * Y8
890 X5 = E1 + X6 * XX + X7 * YY:Y5 =
F1 + Y7 * YY:Z5 = G1 + Z6 * XX + Z7 * YY
900 D = D /  SQR (X5 * X5 + Y5 * Y5 +
Z5 * Z5)
910 NP = NP + 1
920 A(NP,1) = XE + X5 * D
930 A(NP,2) = YE + Y5 * D
940 A(NP,3) = ZE + Z5 * D
950 C%(NP) = CC%:B%(NP,1) = X:B%(NP,2)
= Y:O%(NP) = 0
960 J = NP: IF NP = MP THEN  GOTO 8000
970  IF CC% < 0 OR O%(NP - 1) = 1 THEN
GOTO 1000
980  HCOLOR= CC%:OX = B%(NP - 1,1):
OY = B%(NP - 1,2):F = 1: GOSUB 6000
990 X = B%(NP,1):Y = B%(NP,2)
```

```
1000  GOTO 240
1010  REM
1020  REM  CHANGE COLOUR
1030  FOR I = 1 TO 30: PRINT : NEXT I
1040  VTAB 1
1050  POKE  - 16300,0: POKE  - 16303,0
1060  PRINT "CURRENT COLOUR IS ";
1070  IF CC% < 0 THEN
PRINT "'INVISIBLE'": GOTO 1090
1080  PRINT CC%
1090  PRINT : INPUT "NEW COLOUR (-1 FOR
'INVISIBLE') ";CC%
1100  POKE  - 16299,0: POKE  - 16304,0
1110  GOTO 220
1120  STOP
1130  REM
1140  REM  ROUTINE TO PUT VIEW ONTO
SCREEN
1150 X8 =  COS (TS) *  COS (PS)
1160 Y8 =  SIN (PS) *  COS (TS)
1170 Z8 =  SIN (TS) *  COS (PS)
1180 D1 =  SQR (X8 * X8 + Y8 * Y8 + Z8 * Z8)
1190 X8 = X8 / D1:Y8 = Y8 / D1:Z8 = Z8 / D1
1200 E1 = X8 * R:F1 = Y8 * R:G1 = Z8 * R
1210  FOR I = 1 TO NP
1220 O%(I) = 0
1230 X3 = A(I,1) - XE:Y3 = A(I,2) -
YE:Z3 = A(I,3) - ZE
1240 D =  SQR (X3 * X3 + Y3 * Y3 + Z3 * Z3)
1250 X3 = X3 / D:Y3 = Y3 / D:Z3 = Z3 / D
1260 CA = X8 * X3 + Y8 * Y3 + Z8 * Z3
1270  IF CA < 0 THEN O%(I) = 1
1280 D2 = R / CA
1290 X4 =  - E1 + D2 * X3:Y4 =  - F1 +
D2 * Y3:Z4 =  - G1 + D2 * Z3
1300 X =  COS (TS + PI / 2) * X4 +
SIN (TS + PI / 2) * Z4
1310 SS =  SGN (Y4): IF SS = 0 THEN SS =
-  SGN (X4)
1320 Y =  SQR (X4 * X4 + Y4 * Y4 +
Z4 * Z4 - X * X + 1E - 5) * SS
```

```
1330  IF X > 32000 THEN X = 32000
1340  IF X <  - 32000 THEN X =  - 32000
1350  IF Y > 32000 THEN Y = 32000
1360  IF Y <  - 32000 THEN Y =  - 32000
1370 B%(I,1) = 140 + X:B%(I,2) = 80 + Y
1380  NEXT I
1390  HGR2
1400 F = 0
1410 X = B%(1,1):Y = B%(1,2)
1420  GOSUB 6000
1430  IF NP = 1 THEN  RETURN
1440  FOR I = 2 TO NP
1450 F = 1
1460  IF O%(I) = 1 OR O%(I - 1) = 1 OR
C%(I) < 0 THEN F = 0
1470  IF F = 1 THEN  HCOLOR= C%(I)
1480 X = B%(I,1):Y = B%(I,2)
1490  GOSUB 6000
1500  NEXT I
1510  RETURN
1520  REM
1530  REM  ROUTINE TO PUT CURRENT
POSITION ONTO SCREEN
1540 XL = A(1,1):XS = XL
1550 YL = A(1,2):YS = YL
1560 ZL = A(1,3):ZS = ZL
1570  FOR I = 2 TO NP
1580  IF A(I,1) > XL THEN XL = A(I,1)
1590  IF A(I,1) < XS THEN XS = A(I,1)
1600  IF A(I,3) > ZL THEN ZL = A(I,3)
1610  IF A(I,3) < ZS THEN ZS = A(I,3)
1620  IF A(I,2) > YL THEN YL = A(I,2)
1630  IF A(I,2) < YS THEN YS = A(I,2)
1640  NEXT I
1650  HGR2 : ROT= 0
1660  HCOLOR= 3
1670 K = 0
1680 XD = (XL - XS) / 2:YD = (YL - YS) / 2:
ZD = (ZL - ZS) / 2
1690 M = XD
1700  IF YD > M THEN M = YD
```

```
1710  IF ZD > M THEN M = ZD
1720  IF XE - XS > 2 * M THEN M =
(XE - XS) / 2
1730  IF XL - XE > 2 * M THEN M =
(XL - XE) / 2
1740  IF YE - YS > 2 * M THEN M =
(YE - YS) / 2
1750  IF YL - YE > 2 * M THEN M =
(YL - YE) / 2
1760  IF ZE - ZS > 2 * M THEN M =
(ZE - ZS) / 2
1770  IF ZL - ZE > 2 * M THEN M =
(ZL - ZE) / 2
1780 M = 20 / M
1790 XD = XD * M:YD = YD * M:ZD = ZD * M
1800  HPLOT 140 - XD,50 - YD TO 140 +
XD,50 - YD TO 140 + XD,50 + YD TO 140 -
XD,50 + YD TO 140 - XD,50 - YD
1810  HPLOT 140 - XD,130 - ZD TO 140 +
XD,130 - ZD TO 140 + XD,130 + ZD TO 140
- XD,130 + ZD TO 140 - XD,130 - ZD
1820 X = 140 - XD + (XE - XS) * M:Y = 50
- YD + (YE - YS) * M
1830 S =  ATN (100 / R)
1840  HPLOT X,Y TO X +  COS (PS + S) * 10,
Y +  SIN (PS + S) * 10
1850  HPLOT X,Y TO X +  COS (PS - S) * 10,
Y +  SIN (PS - S) * 10
1860  HPLOT X,Y TO X +  COS (PS) * 10,Y
+  SIN (PS) * 10
1870 Y = 130 - ZD + (ZE - ZS) * M
1880  HPLOT X,Y TO X +  COS (TS + S) * 10,
Y +  SIN (TS + S) * 10
1890  HPLOT X,Y TO X +  COS (TS - S) * 10,
Y +  SIN (TS - S) * 10
1900  HPLOT X,Y TO X +  COS (TS) * 10,
Y +  SIN (TS) * 10
1910  HPLOT 90,90 TO 190,90
1920  SCALE= 2
1930  DRAW 2 AT 20,20: DRAW 1 AT 20,100
1940  SCALE= 1
```

```
1950  DRAW 3 AT 140 - XD + (XE - XS) * M,
  50 - YD + (YE - YS) * M
1960  DRAW 3 AT 140 - XD + (XE - XS) * M,
 130 - ZD + (ZE - ZS) * M
1970  GET A$
1980  IF A$ = "V" THEN  RETURN
1990  IF A$ = "H" THEN XE = XE - 2 / M
2000  IF A$ = "K" THEN XE = XE + 2 / M
2010  IF A$ = "U" THEN YE = YE - 2 / M
2020  IF A$ = "M" THEN YE = YE + 2 / M
2030  IF A$ = "I" THEN ZE = ZE - 2 / M
2040  IF A$ = "N" THEN ZE = ZE + 2 / M
2050 K = K + 1
2060  IF (A$ = "U" OR A$ = "K" OR A$
= "I" OR A$ = "H" OR A$ = "M" OR A$
= "N") AND K < 7 THEN  GOTO 1940
2070  IF A$ = "R" THEN PS = PS +
ATN (200 / R) / 4
2080  IF A$ = "E" THEN PS = PS -
ATN (200 / R) / 4
2090  IF A$ = "B" THEN R = R / 1.5
2100  IF A$ = "S" THEN R = R * 1.5
2110  IF A$ = "F" THEN TS = TS +
ATN (200 / R) / 4
2120  IF A$ = "D" THEN TS = TS -
ATN (200 / R) / 4
2130  IF A$ = " " THEN  GOTO 1650
2140  IF A$ = "X" OR A$ = "Y" OR A$
= "Z" THEN  GOTO 2180
2150  IF A$ = "O" THEN XE = OP(2):
YE = OP(3):ZE = OP(4):TS = OP(5):
PS = OP(6):R = OP(7): GOTO 1650
2160  IF K < 7 THEN  GOTO 1800
2170  GOTO 1650
2180 MM = XD
2190  IF YD > MM THEN MM = YD
2200  IF ZD > MM THEN MM = ZD
2210 MM = MM / M
2220 D = 200 * MM
2230  IF D = 0 THEN D = 2000
```

```
2240 OP(2) = XE:OP(3) = YE:OP(4) = ZE:
OP(5) = TS:OP(6) = PS:OP(7) = R
2250 R = 12000
2260 IF A$ = "X" THEN XE =  - D:YE = 0:
ZE = 0:TS = 0:PS = 0
2270 IF A$ = "Y" THEN XE = 0:YE =  - D:
ZE = 0:TS = 0:PS = PI / 2
2280 IF A$ = "Z" THEN XE = 0:YE = 0:
ZE =  - D:TS = PI / 2:PS = 0
2290  GOTO 1650
2300  REM
5000  HIMEM: 38310
5010  POKE 232,167: POKE 233,149
5020  FOR I = 1 TO 88
5030  READ N
5040  POKE 38310 + I,N
5050  NEXT I
5060  DATA  3,0,66,0,24,0,8,0,27,35
5070  DATA  37,37,45,46,46,54,55,55,63,60
5080  DATA  60,76,1,0,36,36,44,173,26,191
5090  DATA  82,73,36,36,44,45,54,63,15,46
5100  DATA  46,14,36,36,44,45,54,54,62,127
5110  DATA  73,36,36,44,54,53,46,70,35,36
5120  DATA  12,45,51,54,54,0,36,36,28,109
5130  DATA  54,54,46,45,36,36,60,127,73,45
5140  DATA  53,54,63,39,164,18,54,0
5150  RETURN
5160  REM
6000  REM  ROUTINE TO HPLOT OVER THE
SCREEN BORDER
6010  IF F = 0 THEN OX = X:OY = Y:
RETURN : REM  FIRST POINT IN FIGURE
6020  IF OX > X THEN  GOTO 6050
6030 X1 = OX:Y1 = OY:X2 = X:Y2 = Y
6040  GOTO 6060
6050 X1 = X:Y1 = Y:X2 = OX:Y2 = OY
6060 OX = X:OY = Y
6065  IF X1 > 1 AND X1 < X9 AND X2 > 1
AND X2 < X9 AND Y1 > 1 AND Y1 < Y9 AND
Y2 > 1 AND Y2 < Y9 THEN  HPLOT X1,Y1 TO
X2,Y2: RETURN
```

```
6070  IF Y2 = Y1 THEN Y2 = Y1 + P1
6080  IF X2 = X1 THEN X2 = X1 + P1
6090 G = (Y2 - Y1) / (X2 - X1)
6100 X0 = X1 - Y1 / G
6110 N = 0
6120 Y = G * (1 - X0)
6130  IF Y < N1 OR Y > Y9 THEN
GOTO 6150
6140 N = N + N1:P(N) = N1:Q(N) = Y
6150 Y = G * (X9 - X0)
6160  IF Y < N1 OR Y > Y9 THEN
GOTO 6180
6170 N = N + N1:P(N) = X9:Q(N) = Y
6180  IF N = N2 THEN  GOTO 6290
6190  IF N = N1 AND Q(N) = N1 THEN
GOTO 6230
6200 X = X0 + N1 / G
6210  IF X < N1 OR X > X9 THEN
GOTO 6230
6220 N = N + N1:P(N) = X:Q(N) = N1
6230  IF N = N2 THEN  GOTO 6290
6240  IF N = 1 AND Q(N) = Y9 THEN
GOTO 6280
6250 X = X0 + Y9 / G
6260  IF X < N1 OR X > X9 THEN
GOTO 6280
6270 N = N + N1:P(N) = X:Q(N) = Y9
6280  IF N < N2 THEN  RETURN
6290 N = N2
6300  IF P(N1) < P(N2) THEN N = N1
6310 NN = 3 - N
6320  IF X1 > P(NN) OR X2 < P(N) THEN
RETURN
6330  IF X1 >  = P(N) THEN P(N) = X1:
Q(N) = Y1
6340  IF X2 <  = P(NN) THEN P(NN) = X2:
Q(NN) = Y2
6350  HPLOT P(N),Q(N) TO P(NN),Q(NN)
6360  RETURN
7000  REM
```

```
7010  REM  ROUTINE TO READ FROM DISC,
TAPE OR KEYBOARD
7020  TEXT : HOME
7030  PRINT : PRINT : PRINT
7040  PRINT "T = INPUT FROM TAPE"
7050  PRINT "K = INPUT FROM KEYBOARD"
7060  PRINT "D = INPUT FROM DISC"
7070  PRINT
7080  INPUT A$
7090  IF A$ = "D" THEN  GOTO 7300
7100  IF A$ = "T" THEN  GOTO 7530
7110  IF A$ = "K" THEN  GOTO 7140
7120  GOTO 7020
7130  REM
7140  REM  READ FROM SCREEN
7150  PRINT : PRINT "INPUT IN ORDER
X,Y,Z, COLOUR"
7160  PRINT "LAST ENTRY SHOULD BE 9,9,9,9"
7170 NP = 1
7180  INPUT "X,Y,Z,C ";A(NP,1),A(NP,2),
A(NP,3),C%(NP)
7190  IF C%(NP) > 7 THEN  GOTO 7230
7200 NP = NP + 1
7210  IF NP = MP THEN  GOTO 7230
7220  GOTO 7180
7230 NP = NP - 1
7240  INPUT "X,Y,Z OF VIEW ? ";
PP(2),PP(3),PP(4)
7250  INPUT "HORIZ AND VERT ANGLE  ? ";
PP(5),PP(6)
7260  INPUT "MAGNIFICATION ? ";PP(7)
7270 PP(1) = NP
7280  RETURN
7290  REM
7300  REM  INPUT FROM DISC
7310  HOME : PRINT : PRINT
7320  PRINT
7330  PRINT "FILE NAME (* TO SEE
CATALOG) ";
7340  INPUT I$
```

```
7350  IF I$ < > "*" THEN  GOTO 7380
7360  PRINT : PRINT  CHR$ (4);"CATALOG"
7370  GOTO 7320
7380  PRINT  CHR$ (4);"OPEN ";I$
7390  PRINT  CHR$ (4);"READ ";I$
7400  FOR I = 1 TO 7
7410  INPUT PP(I)
7420  NEXT I
7430 NP =  INT (PP(1))
7440  FOR I = 1 TO NP
7450  INPUT A(I,1)
7460  INPUT A(I,2)
7470  INPUT A(I,3)
7480  INPUT C%(I)
7490  NEXT I
7500  PRINT  CHR$ (4);"CLOSE ";I$
7510  RETURN
7520  REM
7530  REM  INPUT FROM TAPE
7540  PRINT : INPUT "GET TAPE READY,
THEN PRESS RETURN ON COMPUTER, THEN PLAY
ON TAPE RECORDER ";A$
7550  RECALL PP
7560  RECALL A
7570  RECALL C%
7580  RETURN
8000  REM  SAVE CURRENT SHAPE
8010  TEXT : HOME
8020 PP(1) = NP:PP(2) = XE:PP(3) = YE:
PP(4) = ZE
8030 PP(5) = TS:PP(6) = PS:PP(7) = R
8040  PRINT : PRINT : PRINT
8050  PRINT "T = SAVE TO TAPE
8060  PRINT "S = SAVE TO SCREEN
8070  PRINT "D = SAVE TO DISC
8080  PRINT : PRINT "C = CANCEL SAVE
8090  PRINT "Q = QUIT (LOSE CURRENT
FIGURE)
8100  PRINT : INPUT A$
8110  IF A$ = "C" THEN  GOTO 180
8120  IF A$ = "Q" THEN  STOP
```

```
8130  IF A$ = "T" THEN  GOTO 8290
8140  IF A$ = "S" THEN  GOTO 8170
8150  IF A$ = "D" THEN  GOTO 8380
8160  GOTO 8010
8170  HOME
8180  PRINT : PRINT "X,Y,Z, COLOUR:"
8190  FOR I = 1 TO NP
8200  PRINT A(I,1),A(I,2),A(I,3),C%(I)
8210  NEXT I
8220  PRINT : PRINT "NUMBER OF POINTS = ";NP
8230  PRINT "X,Y,Z OF VIEW = ";XE,YE,ZE
8240  PRINT "HORIZ AND VERT ANGLES = ";
TS,PS
8250  PRINT "MAGNIFICATION = ";R
8260  INPUT A$
8270  GOTO 180
8280  REM
8290  REM  SAVE TO TAPE
8300  TEXT : HOME
8310  INPUT "SET TAPE TO RECORD, THEN
PRESS RETURN ";A$
8320  PRINT "COMPUTER WILL GIVE SIX
BEEPS"
8330  STORE PP
8340  STORE A
8350  STORE C%
8360  GOTO 180
8370  REM
8380  REM  SAVE TO DISC
8390  TEXT : HOME
8400  PRINT "FILE NAME (* TO SEE
CATALOG) ";
8410  INPUT O$
8420  IF O$ < > "*" THEN  GOTO 8450
8430  PRINT : PRINT  CHR$ (4);"CATALOG"
8440  PRINT : GOTO 8400
8450  PRINT  CHR$ (4);"OPEN ";O$
8460  PRINT  CHR$ (4);"WRITE ";O$
8470  FOR I = 1 TO 7
8480  PRINT PP(I)
8490  NEXT I
```

```
8500  FOR I = 1 TO PP(1)
8510  PRINT A(I,1): PRINT A(I,2):
PRINT A(I,3): PRINT C%(I)
8520  NEXT I
8530  PRINT  CHR$ (4);"CLOSE ";O$
8540  GOTO 180
```

Lines 1 and 2 are part of the EXPLOT routine discussed earlier.

Line 10 defines PI (the number of radians in 180 degrees) which is used frequently in the program. It also sets CC%, which is the current color.

Line 20 performs the subroutine at line 5000, which sets up the various shapes in memory that we will need: the cursor (which is also used as the 'point of view' in the map), and the words FRONT and TOP, also for the map.

Line 30 sets HIMEM: to the bottom of the high-resolution screen. This is to stop the program encroaching (if you modify it later) on the video display area. The program is so large (mainly due to the arrays, and number of lines of program) that it runs a risk of overlapping the video area if it gets much bigger. If you *are* modifying the program and you need more room, remember that you can always decrease the size of the arrays (line 40) from 150 elements (i.e. 150 points) to a smaller number. The value of MP will change, too, as it shows the maximum number of points that the program can handle.

Lines 50 to 90 find out whether you want a new figure, or an old one.

Lines 100 to 120 set up the figure storage parts of the program for a 'null' figure. If you have asked for a 'new'

figure, this will be what is in the machine when you start.

NP is the number of points. XE, YE and ZE hold the co-ordinates of the point of view. PS and TS are the angles that define the direction of view.

A(,) is the array that holds the co-ordinates of the points in the figure. C%() is the array that holds the color of each point.

J is the variable that holds the number of the point that was last 'marked', either by the use of the P and Q keys, or by the use of the J key.

Line 140 sends the program to the routine at line 7000, which inputs a figure from the keyboard, tape or disc. This routine is fairly straightforward with the possible exception of lines 7330 to 7370, which ask for the name of the disc file to be read. If you enter an asterisk (*), then you will get the current disc CATALOG. With this method, unless you actually want the CATALOG, you only need to enter the file name. The tape reading part of the routine at lines 7550 to 7570, reads in the three arrays that hold (in order) the current point of view and direction of view, etc. (this is PP()), the co-ordinates of the points in the figure (A(,)) and the color of each (C%()).

At line 150, the current values of NP, XE, YE, ZE, TS, PS and R (the magnification) are stored in array PP() for use when we decide to save this figure.

The subroutine at line 1140, which puts the current view onto the screen, is the one we have just examined, so we won't go into it again.

X and Y are used to store the current position of the cursor on the screen. Line 190 sets this to the position of the last point 'marked' (i.e. the point whose number is in J).

Lines 200 and 210 make sure that the cursor stays on the screen, even if the marked point is *off* the screen. Line 240 actually draws the cursor on the screen.

Line 250 is a pause until the next key is pressed. When this happens, line 260 erases the cursor ready to re-draw it in its new position after we have decided what to do about that key press.

Lines 270 to 310 deal with cursor movement.

Lines 320 to 370 deal with the various other function keys.

Programming style

Notice that, in contrast to accepted good programming methods, I haven't used GOSUB and RETURN to process these function keys. Try writing a program segment to do the task that we need to do here and you'll see why. In BASIC, you RETURN to the line *after* the GOSUB. So, in this application you would have no choice about how far back up the loop you go after you execute the routine. In some instances, we want to re-draw the screen. In others, we just want to move the cursor. Using GOTOs means that we have more control over where the program goes (although admittedly it does lead, in general, to more programming mistakes).

Lines 380 to 400 control the cursor position in terms of the P and Q keys (which move the cursor to the 'previous' and 'next' points of the figure), and the Z key (which moves it to the 'last' point of the figure).

Line 410 erases the last added point simply by reducing NP, the number of points in the figure.

The routine at line 480 is the one operated by the J key. This 'marks' the point that the cursor is over by comparing the current cursor position with the contents of array B%(,), which, as you will remember from the analysis of the routine that puts the view onto the screen, holds the screen co-ordinates of each of the points of the figure. If there is no point that agrees (i.e. if the cursor is not directly over one of the points), then the computer will give a series of three beeps (lines to 560).

Line 570 sends the program back to get the next key press (without moving anything on the screen).

The routine at line 590 is more complex. It is operated by the F key, and takes the last marked point (i.e. the one whose number is contained in variable J) and moves it to the current cursor position. It will also change the color of the line associated with the point to whatever is in CC% (the 'current color' variable).

Firstly, line 610 saves the current cursor position in XX and YY (since we'll need to use X and Y to communicate with our EXPLOT routine later). Then line 620 sets D to the distance of the figure point from the point of view. In effect, we move the figure point as the cursor has moved on the screen, while *keeping the point at the same distance from the point of view.*

Line 630 sets vector 6 (X6, Y6, Z6) to the value of a unit vector pointing to the right across the screen (Y6, which will always be zero since this line is horizontal, is left out).

Then lines 640 to 660 calculate vector 7, which is a vector in the direction of 'up the screen', by taking vector 6, and vector 8 (the direction of view), and using them to find a vector which is at right angles to both of them.

Line 670 calculates vector 5, the vector from the point of view to the present cursor position. This is done

by multiplying the unit vector which points *across* the screen (vector 6) by the *distance* which the cursor is *across* the screen (XX); and then multiplying the up-screen unit vector (vector 7) by the up-screen distance (YY), adding the two together to give the vector from the middle of the screen to the cursor, and then adding the vector from the point of view to the middle.

Line 680 takes the current value of D (the distance of the marked point from the point of view) and divides it by the length of the vector from the point of view to the cursor.

At lines 690 to 710, the new position is determined by multiplying vector 5 by the new value of D. If you think about this, you will see that multiplying vector 5 by D will make it a vector pointing from the point of view to the cursor, but with a length which is equal to the distance from the point of view to the old position of the point that is being moved.

Line 720 sets the color of the point, and sets O%() to 0, which marks the point as being within the field of view.

Line 730 sets B%(,) to the new position of the point on the screen, and sets F to 1 for the purpose of communicating with the EXPLOT routine, later.

It now remains to draw in the lines that start and end at the point just moved in their new positions.

Firstly, we deal with the line that finishes at the point. 740 takes care of the case where the color of that line is 'invisible', or where the point that it starts at is out of the field of view. In either case the line is not to be drawn. Then line 750 does an EXPLOT from the previous point (point number J minus 1) to the new position of the moved point.

Secondly, we draw the line that *starts* at the moved point. Line 760 takes care of the case where there isn't one (i.e. point J is the last point). Line 770 takes care of the case where the line is 'invisible', or goes out of the field of view. Line 780 does the EXPLOT, line 785 sets X and Y back to where they should be, and then line 790 sends the program back for the next key press.

The routine that adds a point begins at line 810, and is very similar to the last routine (the one to move a point).

Changing color

The 'change color' routine begins at line 1020. This is made possible by the Apple's capability to move its displayed screen from one part of memory to another. To print text onto the screen without using this facility, it is necessary to erase the screen, and go to TEXT mode. Also, to get the view back onto the screen the program must re-draw the screen image, and this usually takes quite a few seconds. As the C key is used often in drawing a figure on the screen (mainly to go to and from the 'invisible' mode), this would have slowed the operation of the program down considerably.

In this routine, line 1030 clears the text screen, and line 1040 puts the cursor (the text screen cursor — nothing to do with the figure-drawing one) to the top of the screen. Then line 1050 does a couple of POKEs that change the display from the second HGR screen (the one

we're using for the view) to the TEXT screen without disruptions to the HGR screen contents.

Lines 1060 to 1090 assign the new color, and finally line 1100 takes us back to HGR2 with the view still intact.

Next is the routine which has already been explained that puts the view onto the screen.

At line 1530 begins the other major routine which puts the 'map' up on the screen.

This routine consists of three parts: the 'fixed' information (the words FRONT and TOP, and the line across the screen); the two cursors that show the point and direction of view; and the two boxes that show the outline of the figure. The boxes fit round the figure so that the left-hand edge, for example, is at the position of the lowest X-co-ordinate of the figure.

The screen display is assembled in such a way that the size of the boxes and the positions of the point of view cursors are always in scale.

Firstly, we need to find the maximum and minimum value of each co-ordinate (X, Y and Z) for the points in the figure.

Lines 1540 to 1560 set up six variables — XL for 'lowest X', XH for 'highest X', and so on — to starting values. These starting values are the co-ordinates of the first point in the figure.

The loop from 1570 to 1640 compares the co-ordinates of each of the points of the figure with the high and low values, changing them as appropriate. By the time we get to line 1650, XL, XH, YL, YH, ZL and ZH are at the highest and lowest values and determine the edges of the boxes.

Line 1650 clears the screen, and line 1660 sets the color to white.

Line 1670 sets K to zero. This will be the 'counter' variable that counts the number of key presses that have been made without the screen being re-drawn, and automatically re-draws the screen if it reaches 7.

Line 1680 finds XD, YD and ZD representing the widths of the box around the figure, divided by 2. So, from the centre of the box, we would go XD, YD or ZD to get to the side of the box.

Line 1690 sets M to XD, and then lines 1700 and 1710 set it to the maximum of either XD, YD or ZD. M is now the maximum dimension of the box (divided by 2).

Lines 1720 to 1770 decide whether the point of view is beyond this maximum distance (M) from the middle of the box, in which case M is updated accordingly. M is now the maximum distance of *anything* from the middle of the box in the X, Y and Z directions. Line 1780 changes M from a maximum distance to a number with which we can multiply the distances involved to give products of 20 or less.

Scale

The variable M represents the *scale* of the display. We can multiply everything by M and know that when we put it on the screen, nothing will go beyond the edge.

Line 1790 converts XD, YD and ZD to their scale values.

Lines 1800 and 1810 draw the boxes onto the screen.

Then line 1820 works out the position for the FRONT view of the point of view.

Line 1830 works out the correct angle for the 'width of screen' lines which are to go onto the point of view cursors.

Then lines 1840 to 1860 put the 'direction of view' and 'width of screen' lines onto the FRONT cursor (which has not yet been drawn on the screen).

Lines 1880 to 1900 do the same for the TOP point of view cursor. The reason for drawing these direction and width lines on the screen before we draw the cursor will become apparent as we proceed.

Line 1910 puts the horizontal line on the screen, and line 1930 prints the words FRONT and TOP. Then lines 1950 and 1960 draw the diamond-shaped point of view cursors. This is done last so that we can loop back to line 1940 when moving the cursor across the screen, and thus avoid having to re-draw the width and direction lines when moving the point of view. In this way we save computing time and speed-up the cursor movement. So, when the REPT key is used with the U, I, K, M, N or H keys (the ones that move the point of view), the cursor moves quite quickly across the screen, stopping at every seventh move (when the K variable reaches 7) for the screen to be re-drawn automatically. This is so that the cursors cannot 'fall off the edge of the screen'. Re-drawing the screen changes the scale, so they never reach the edge.

Lines 1990 to 2040 update the point of view. Line 2050 increments K (i.e. 'adds one to it').

Line 2060 sends the program round the loop again, without checking for other keys, if all we have done is to move the point of view.

Lines 2070 and 2080 rotate the FRONT view of the

direction of view by an amount which depends on the screen width. This means that if you want to move it around quickly, you have to change the width. It also means that when you have the figure almost on the screen, you know that by rotating the direction of view by one key press, you will have moved the figure by the same amount across the screen no matter what the screen width is.

Lines 2090 and 2100 change the magnification (R) and therefore the 'screen width'.

Lines 2110 and 2120 deal with the rotation of the TOP direction of view.

Line 2130 re-draws the screen if you press the space bar.

Line 2140 sends the program out of the loop if you press the X, Y or Z keys — these do something special.

Line 2150 deals with the O key by re-instating the last-saved point, direction of view, and magnification. If K is less than 7, then you go back round the loop from line 2160. However, if you have pressed a key seven times in a row without re-drawing the screen, you will drop through line 2160, and line 2170 sends the program further back up the loop, to re-draw the screen.

Getting to line 2180 (which you would if you had pressed X, Y or Z), MM is set to the maximum of XD, YD or ZD and it now represents the scaled largest dimension of the box, divided by two.

Line 2210 divides it by M, giving not the scaled version of this largest dimension (remember, XD, YD and ZD are scaled), but the actual largest dimension of the box (divided by two).

Line 2220 calculates D, the distance *away* from the figure that we will go to give our 'perspective-free' view. This assumes, by the way, that the figure is centred around the point with co-ordinates 0, 0, 0. If this is not the case, then you will get an off-centre image when you use the X, Y or Z keys. It is best to centre the figure roughly about 0, 0, 0, the position of the cursor when you start the program.

Line 2230 sets D to an arbitrary value if there is no figure in memory, as is the case when you first start the program up.

Line 2240 saves the old position, so that it can be recalled using the O key.

Lines 2260 to 2280 set up the required values for a long-distance view from one of the three directions X, Y or Z. Then line 2290 does the loop again, re-drawing the screen.

The routine at line 8000 saves the current figure to screen, tape or disc.

Animation

An obvious application of the THREE-D FIGURE program is in animation. We can design a figure, look at it from different points of view, and move the points in the figure at the same time.

Included in this section is a program that will automatically transform one three-dimensional figure into another in equal stages. A listing of the program is given here. To operate the program, RUN it, and then

input the sources of the two figures to be used, remembering that they have to have the same number of points. Lastly, specify the number of steps (the number of views that will be generated will be this number of steps plus one).

MOVE THREE-D FIGURE

```
1  DIM P(2),Q(2):X9 = 279:Y9 = 191
2 P1 = 0.1:N1 = 1:N2 = 2
10 PI = 3.14159265:CC% = 3
15  HIMEM: 16384
20  DIM A(150,3),B%(150,2),O%(151),
C%(150),PP(7)
30  DIM A1(150,3),V1(7),V2(7)
40  TEXT : HOME
50  PRINT "FIRST SHAPE": PRINT : PRINT
60  GOSUB 7040
70  FOR I = 1 TO 7
80 V1(I) = PP(I)
90  NEXT I
100  FOR I = 1 TO PP(1)
110 A1(I,1) = A(I,1):A1(I,2) = A(I,2):
A1(I,3) = A(I,3)
111  NEXT I
115  HOME
120  PRINT "SECOND SHAPE": PRINT : PRINT
130  GOSUB 7040
140  FOR I = 1 TO 7
150 V2(I) = PP(I)
160  NEXT I
170  IF V2(1) <  > V1(1) THEN
PRINT "SHAPE 1 HAS ";V1(1);" POINTS AND
SHAPE 2 HAS ";V2(1): STOP
180 NP =  INT (V2(1))
210  TEXT : HOME
```

```
220  PRINT "HOW MANY STEPS ";
225  INPUT NS
230  FOR I = 1 TO NP
231  FOR J = 1 TO 3
232 A = A1(I,J):A1(I,J) = (A(I,J) -
A1(I,J)) / NS:A(I,J) = A - A1(I,J)
233  NEXT J
235  NEXT I
240  FOR N9 = 0 TO NS
250 W1 = (NS - N9) / NS:W2 = N9 / NS
260 XE = W1 * V1(2) + W2 * V2(2)
270 YE = W1 * V1(3) + W2 * V2(3)
280 ZE = W1 * V1(4) + W2 * V2(4)
290 TS = W1 * V1(5) + W2 * V2(5)
300 PS = W1 * V1(6) + W2 * V2(6)
310 R = W1 * V1(7) + W2 * V2(7)
330  FOR I = 1 TO NP
340 A(I,1) = A(I,1) + A1(I,1)
350 A(I,2) = A(I,2) + A1(I,2)
360 A(I,3) = A(I,3) + A1(I,3)
370  NEXT I
380  GOSUB 1140
390  INPUT A$
395  TEXT : HOME
396  PRINT "PROCESSING STEP ";N9 + 1
400  NEXT N9
410  TEXT : HOME : STOP
1140  REM  ROUTINE TO PUT VIEW ONTO
SCREEN
1150 X8 =  COS (TS) *  COS (PS)
1160 Y8 =  SIN (PS) *  COS (TS)
1170 Z8 =  SIN (TS) *  COS (PS)
1180 D1 =  SQR (X8 * X8 + Y8 * Y8 + Z8 * Z8)
1190 X8 = X8 / D1:Y8 = Y8 / D1:Z8 = Z8 / D1
1200 E1 = X8 * R:F1 = Y8 * R:G1 = Z8 * R
1210  FOR I = 1 TO NP
1220 O%(I) = 0
1230 X3 = A(I,1) - XE:Y3 = A(I,2) - YE:
Z3 = A(I,3) - ZE
1240 D =  SQR (X3 * X3 + Y3 * Y3 + Z3 * Z3)
1250 X3 = X3 / D:Y3 = Y3 / D:Z3 = Z3 / D
```

```
1260 CA = X8 * X3 + Y8 * Y3 + Z8 * Z3
1270  IF CA < 0 THEN O%(I) = 1
1280 D2 = R / CA
1290 X4 =  - E1 + D2 * X3:Y4 =  - F1 +
D2 * Y3:Z4 =  - G1 + D2 * Z3
1300 X =  COS (TS + PI / 2) * X4 +
SIN (TS + PI / 2) * Z4
1310 SS =  SGN (Y4): IF SS = 0 THEN SS =
-  SGN (X4)
1320 Y =  SQR (X4 * X4 + Y4 * Y4 +
Z4 * Z4 - X * X) * SS
1330  IF X > 32000 THEN X = 32000
1340  IF X <  - 32000 THEN X =  - 32000
1350  IF Y > 32000 THEN Y = 32000
1360  IF Y <  - 32000 THEN Y =  - 32000
1370 B%(I,1) = 140 + X:B%(I,2) = 80 + Y
1380  NEXT I
1390  HGR2
1400 F = 0
1410 X = B%(1,1):Y = B%(1,2)
1420  GOSUB 6000
1430  IF NP = 1 THEN  RETURN
1440  FOR I = 2 TO NP
1450 F = 1
1460  IF O%(I) = 1 OR O%(I - 1) = 1 OR
O%(I + 1) = 1 OR C%(I) < 0 THEN F = 0
1470  IF F = 1 THEN  HCOLOR= C%(I)
1480 X = B%(I,1):Y = B%(I,2)
1490  GOSUB 6000
1500  NEXT I
1510  RETURN
1520  REM
6000  REM  ROUTINE TO HPLOT OVER THE
SCREEN BORDER
6010  IF F = 0 THEN OX = X:OY = Y:
RETURN : REM  FIRST POINT IN FIGURE
6020  IF OX > X THEN  GOTO 6050
6030 X1 = OX:Y1 = OY:X2 = X:Y2 = Y
6040  GOTO 6060
6050 X1 = X:Y1 = Y:X2 = OX:Y2 = OY
6060 OX = X:OY = Y
```

```
6065  IF X1 > 1 AND X1 < X9 AND X2 > 1
AND X2 < X9 AND Y1 > 1 AND Y1 < Y9 AND
Y2 > 1 AND Y2 < Y9 THEN  HPLOT X1,Y1 TO
X2,Y2: RETURN
6070  IF Y2 = Y1 THEN Y2 = Y1 + P1
6080  IF X2 = X1 THEN X2 = X1 + P1
6090 G = (Y2 - Y1) / (X2 - X1)
6100 X0 = X1 - Y1 / G
6110 N = 0
6120 Y = G * (1 - X0)
6130  IF Y < N1 OR Y > Y9 THEN
GOTO 6150
6140 N = N + N1:P(N) = N1:Q(N) = Y
6150 Y = G * (X9 - X0)
6160  IF Y < N1 OR Y > Y9 THEN
GOTO 6180
6170 N = N + N1:P(N) = X9:Q(N) = Y
6180  IF N = N2 THEN  GOTO 6290
6190  IF N = N1 AND Q(N) = N1 THEN
GOTO 6230
6200 X = X0 + N1 / G
6210  IF X < N1 OR X > X9 THEN
GOTO 6230
6220 N = N + N1:P(N) = X:Q(N) = N1
6230  IF N = N2 THEN  GOTO 6290
6240  IF N = 1 AND Q(N) = Y9 THEN
GOTO 6280
6250 X = X0 + Y9 / G
6260  IF X < N1 OR X > X9 THEN
GOTO 6280
6270 N = N + N1:P(N) = X:Q(N) = Y9
6280  IF N < N2 THEN  RETURN
6290 N = N2
6300  IF P(N1) < P(N2) THEN N = N1
6310 NN = 3 - N
6320  IF X1 > P(NN) OR X2 < P(N) THEN
RETURN
6330  IF X1 >  = P(N) THEN P(N) =
X1:Q(N) = Y1
6340  IF X2 <  = P(NN) THEN P(NN) =
X2:Q(NN) = Y2
```

```
6350  HPLOT P(N),Q(N) TO P(NN),Q(NN)
6360  RETURN
7000  REM
7010  REM  ROUTINE TO READ FROM DISC,
TAPE OR KEYBOARD
7020  TEXT : HOME
7030  PRINT : PRINT : PRINT
7040  PRINT "T = INPUT FROM TAPE"
7050  PRINT "K = INPUT FROM KEYBOARD"
7060  PRINT "D = INPUT FROM DISC"
7070  PRINT
7080  INPUT A$
7090  IF A$ = "D" THEN  GOTO 7300
7100  IF A$ = "T" THEN  GOTO 7530
7110  IF A$ = "K" THEN  GOTO 7140
7120  GOTO 7020
7130  REM
7140  REM  READ FROM SCREEN
7150  PRINT : PRINT "INPUT IN ORDER
X,Y,Z, COLOUR"
7160  PRINT "LAST ENTRY SHOULD BE 9,9,9,9"
7170 NP = 1
7180 INPUT "X,Y,Z,C ";A(NP,1),A(NP,2),
A(NP,3),C%(NP)
7190  IF C%(NP) > 7 THEN  GOTO 7230
7200 NP = NP + 1
7210  IF NP = MP THEN  GOTO 7230
7220  GOTO 7180
7230 NP = NP - 1
7240  INPUT "X,Y,Z OF VIEW ? ";
PP(2),PP(3),PP(4)
7250  INPUT "HORIZ AND VERT ANGLE ? ";
PP(5),PP(6)
7260  INPUT "MAGNIFICATION ? ";PP(7)
7270 PP(1) = NP
7280  RETURN
7290  REM
7300  REM  INPUT FROM DISC
7310  HOME : PRINT : PRINT
7320  PRINT
```

```
7330  PRINT "FILE NAME (* TO SEE
CATALOG) ";
7340  INPUT I$
7350  IF I$ < > "*" THEN  GOTO 7380
7360  PRINT : PRINT  CHR$ (4);"CATALOG"
7370  GOTO 7320
7380  PRINT  CHR$ (4);"OPEN ";I$
7390  PRINT  CHR$ (4);"READ ";I$
7400  FOR I = 1 TO 7
7410  INPUT PP(I)
7420  NEXT I
7430 NP =  INT (PP(1))
7440  FOR I = 1 TO NP
7450  INPUT A(I,1)
7460  INPUT A(I,2)
7470  INPUT A(I,3)
7480  INPUT C%(I)
7490  NEXT I
7500  PRINT  CHR$ (4);"CLOSE ";I$
7510  RETURN
7520  REM
7530  REM  INPUT FROM TAPE
7540  PRINT : INPUT "GET TAPE READY,
THEN PRESS RETURN ON COMPUTER, THEN PLAY
ON TAPE RECORDER ";A$
7550  RECALL PP
7560  RECALL A
7570  RECALL C%
7580  RETURN
```

This program includes large segments of the THREE-D FIGURE program. The part that changes the co-ordinates of each point, of the point of view, and of all of the other parameters of the program have been added. Using this program, you can produce changing views of figures which themselves change (see Figure 3–12).

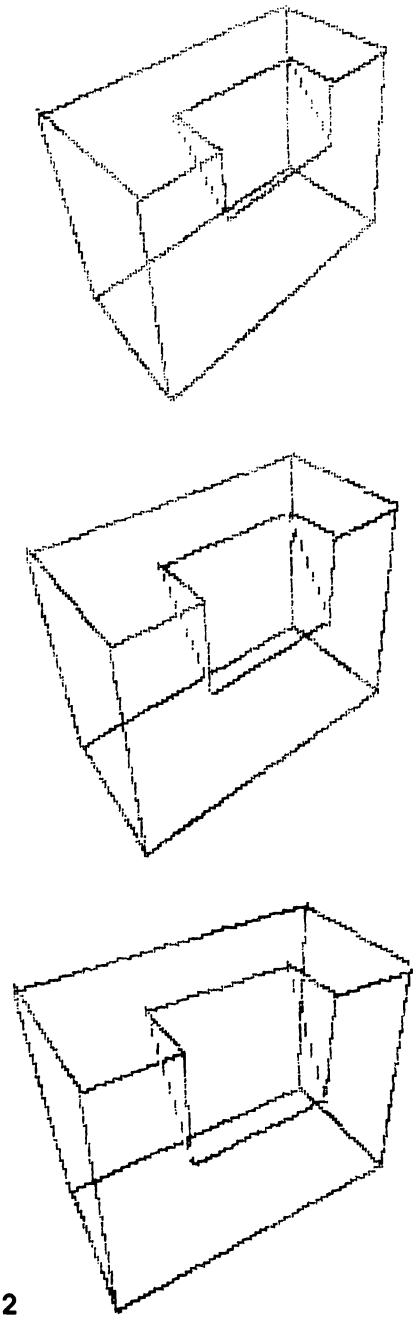
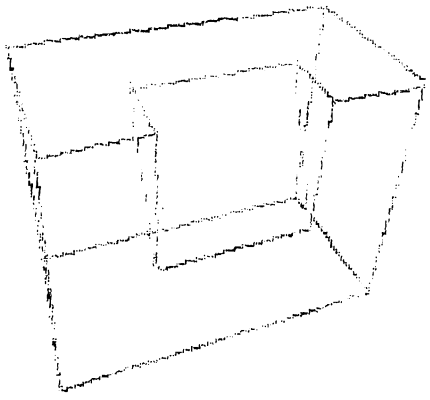


Figure 3–12



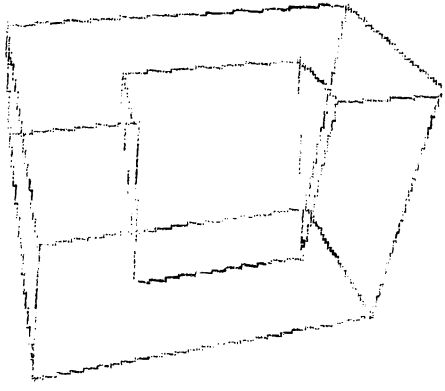
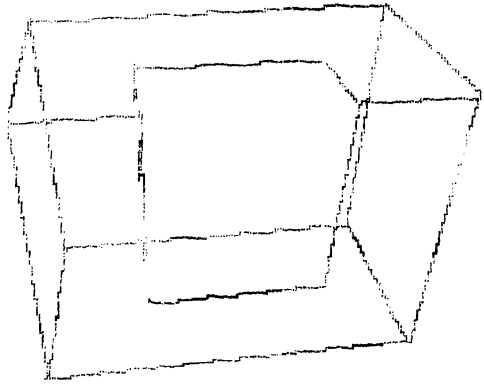


Figure 3–12 (continued)



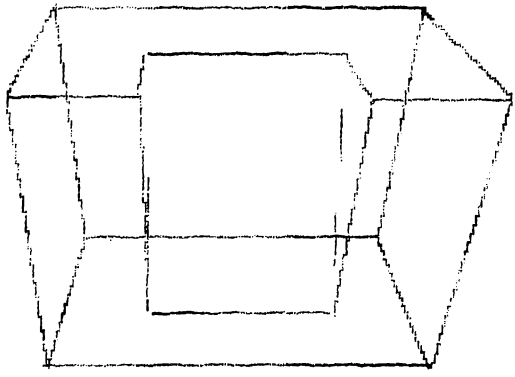


Figure 3–12 (continued)

This program can also be used for technical drawing. It is possible to construct a complete model of a three-dimensional shape, and generate 'views' of the shape by moving the point of view to various positions. For architectural drafting, you may want to use the perspective that the program gives to make the views more realistic. However in other cases, for example drawing machine parts, you will probably want to get rid of the perspective completely. To achieve this, move the point of view as far away from the figure as possible. The X, Y and Z commands in the program do that for you automatically, giving you views which correspond to END ELEVATION, PLAN and ELEVATION respectively.

The program is very flexible and the addition of a printer makes it even more useful. Like any tool, its usefulness is limited only by the imagination of the user.

There are other possibilities worth exploring. For example, those amongst you who are familiar with electronics might like to consider designing a device for feeding shapes directly into the computer. This could consist of an arrangement of arms with potentiometers at their swivel points. Interfacing with the Apple is easy. By monitoring the resistance of each of the potentiometers, the computer can keep track of where the end of the arm arrangement is in space. It would be a simple matter then to translate a series of positions in real space into co-ordinates for the program. You could 'trace' the shape of a real object and get it into the machine that way.

Another possibility is a three-dimensional VDU. By using the machine to generate two views of the same figure, and by displaying them side by side on the screen, you could get the same effect as produced by the old 3-D

viewers. The views would correspond to the position in space of a pair of eyes. All you would then need to do would be to ensure that one of your eyes saw one view, and the other eye saw the other. Your brain would do the rest, to produce a three-dimensional view of the figure. Having done that, it would be pretty simple to write a program that generated a three-dimensional graph or chart.

But these are just the tip of the iceberg — once you have control over a three-dimensional space inside the machine, and once you can move things about in that space and view it from a movable position, then your imagination will generate plenty of ideas for applications.

I hope that this book will provide a jumping-off point for your own projects. I hope that by following the route that I've laid out here, you will find plenty of side-roads that lead into areas even more interesting and even more useful. Good luck and good travelling.

Binary

Appendix 1

The binary numbers system is based on units of 2s rather than units of 10s as used in the normal decimal system.

In the decimal system this is how we represent the number 124:

Hundreds	*Tens*	*Units*
1	2	4

It is made up of one 'hundred', two 'tens' and four 'units' (or 'ones'). The value of each column goes up in multiples of ten. So the first column (the 'units' column) is worth 1, the next is worth 10, the next 100 and so forth.

Notice also that the number of possible entries in each column is 10 (that is, 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 — ten possibilities). So you see, in our 'normal' arithmetic, which is called *base 10* arithmetic, the value of each

column goes up by a factor of 10, and the number of possible entries in each column is 10.

Arithmetic can be done in other base number systems. Computers use a *base 2* or *binary* system.

In computers, numbers are represented using the principle of 'switches' or 'lights' each of which has two possible states — 'on' or 'off'. So in a *base 2* number system, the values of the columns would be multiples of 2 with two possible entries in each column — **0** or **1**. The columns would be:

Eights	*Fours*	*Twos*	*Units*

The decimal number 6 would be represented in base 2 by one 'four' and one 'two':

Fours	*Twos*	*Units*
1	**1**	**0**

We are going to use bold type in this appendix to make the binary figures stand out clearly. Now one 'four' plus one 'two' makes six, and so the number **110** in binary is equal to the number '6' in base 10.

All decimal numbers can be written in binary form. For example, here are the binary equivalents of the numbers from 0 to 15:

Base 10	*Binary equivalent*
0	**0**
1	**1**
2	**10**
3	**11**
4	**100**
5	**101**
6	**110**
7	**111**
8	**1000**
9	**1001**
10	**1010**
11	**1011**
12	**1100**
13	**1101**
14	**1110**
15	**1111**

Use this table to get some practice in going from one system to the other. Also, take a look at how the binary numbers go up from 4. The next binary number is **101** — as in base 10, **101** follows **100**. But the next number is **110**. In binary, you can't have more than **1** in any column, so the next higher number 'overflows' into the column to the left.

There are some special terms used with binary. For example, *bit* is short for 'binary digit'. A 'bit' is a single digit of a binary number. For example, binary **101** is a 'three-bit number'.

Byte is the name given to an 'eight-bit binary number'. Most home computers use eight-bit numbers, and that is why they get a special name.

Because there are only two possible entries in each column of a binary number, we sometimes talk about a particular bit of the number being 'on' (**1**) or 'off' (**0**).

You should now be able to understand this sentence: '**10001001** is a byte which has three bits on and the remainder off.'

The *most-significant* 'bit' is the left-most bit of a binary number, while the *least-significant* bit is the right-most bit. So, if we take the byte **10010010** and turn 'on' the least-significant bit, we get **10010011**.

Here are some useful eight-bit binary numbers that are probably worth remembering.

Base 10	*Binary*
0	**00000000**
15	**00001111**
255	**11111111**

Because we don't usually think in terms of binary numbers, we need to know how to convert from binary to 'decimal' and back again.

Going from binary to base 10 is easy. Let's look at the byte **10101010**. From *right* to *left*, that's **0** units, **1** two, **0** fours, **1** eight, **0** sixteens, **1** thirty-two, **0** sixty-fours and **1** one-hundred-and-twenty-eight. Additing it up gives $2+8+32+128=170$.

Going the other way is only slightly more difficult. For simplicity, let's use only eight-bit binary numbers so the decimal numbers we will be dealing with will be between 0 and 255 (see table above). Take the decimal number 170 for example. Firstly, to see whether the

most-significant bit is going to be on, we subtract 128 from the number (128 is the value of the eighth column from the left in binary). 170−128=42, 42 is the remainder. Now we try taking 64 away from that remainder. It doesn't go, so we try subtracting 32 from 42 which leaves 10. So far we know that 170 is made up of (**1**×128)+(**0**×64)+(**1**×32)+10. The remainder, 10, we can convert using the table near the start of this appendix to give **1010**. This gives us the binary equivalent of 170 which equals:

128s	64s	32s	16s	8s	4s	2s	1s
1	**0**	**1**	**0**	**1**	**0**	**1**	**0**

Notice some interesting things about the binary number system. An *even* number will always have **0** as the least significant bit. If we multiply a binary number by 2, all of the digits will shift one place to the left (just the same as if you multiply a decimal number by 10). Now would be a good time to get a pencil and paper and try some binary arithmetic - you'll soon get the knack of it, and from then on, you'll find binary easier to understand and work with.

Almost all the arithmetic done by a computer is in the binary system (even though all of the results are converted back into decimal so that you and I can understand them). Even the internal structure of the computer is based on the binary system. For example, the number of 'memory locations' (the number of bytes that the computer can hold in its memory) is usually a round number *in binary*. For example, the number of locations

in the Apple's memory is 49152, which is **110000000000** in binary.

Since the number of memory locations is one of the things that people are most interested in when buying a computer, it would be convenient to have a shorthand way of describing this large, round, binary number (which is rather a horrible 49152 in decimal). The letter 'K' (capital K) is used to represent **1000000000** which equals 1024 in decimal. So, the number of memory locations in the Apple is described as being '48K', which is $48 \times 1024 = 49152$.

The number 1024 was chosen because it is a round number in binary, and is roughly equal to 1000 in decimal. Using K, we can specify the number of locations *exactly* (49152) and at the same time say that it's *roughly* 48000 (which makes sense in decimal).

This brief introduction to binary is all you need to understand this book. Remember, though, that a bit of practice will make it easier. What about writing a couple of BASIC programs for your Apple that will convert numbers to and from binary?

Attaching a Printer

Appendix 2

Soon after you start generating computer graphics, you might decide that you need 'hard copy' — a printed copy of whatever is on the screen.

There are a number of methods of getting hard copies of the graphics on the screen. You could place a sheet of tracing paper over the screen of your monitor and trace what is on the screen. Alternatively, you could photograph the image on the screen. When photographing a TV screen, you should use an exposure time between 1/8th and 1/2 second. Any shorter, and you might run into problems with the 'refresh' rate of the TV screen. Any longer than 1/2 second, and the 'wavering' encountered with some TV sets may cause trouble. If you are a keen photographer, you may find this method quite good — beware, though, of the poor 'linearity' of some TV sets. Sometimes the image gets 'squashed' at either the top or bottom of the screen, making accurate images impossible.

By far the best way of capturing your computer's output on paper is by using a printer. There are basically two types of printer available at the domestic end of the computer market: daisy-wheel printers, and dot matrix printers.

Daisy-wheel printers are based on a mechanism that prints letters on paper by bringing a raised, letter-shaped 'petal' up against the paper with an ink ribbon in between.

Dot matrix printers, however, form letters using individual dots placed in precise positions on the paper. Letters are formed by putting the correct pattern of dots onto the paper.

Of these two types of printer, the daisy-wheel, which is the more expensive, gives the better result if only text is to be printed.

For computer graphics, the dot matrix type of printer is necessary. Each 'dot' of the screen becomes one of the 'dots' that the printer makes on the page. This is not possible with *all* dot-matrix printers. Some of them are 'dedicated' to printing letters, and do not have a 'binary' mode in which the computer actually controls the printing dot by dot.

You are lucky to be buying a printer for the Apple computer because the Apple is very widely used, and there are many printers (and other peripherals) available for it off the shelf. Most shops that stock the Apple will carry at least one dot matrix printer that will suit your needs.

When buying a printer, make sure that you actually see the printer being demonstrated properly. Ask the salesman to put something onto the screen in high-resolution graphics mode, and then print it.

Special software is usually provided (sometimes at additional cost) with a printer that has this capability.

If you have a disc drive, there are two ways of getting hard copy (using the printer) of what is on the screen.

You *can* copy the screen contents directly from memory by running the relevant program to get them onto the screen, interrupting the program (with a CTRL-C), and then loading the 'printer driver' software that puts the screen contents onto the printer. However, you may find this tedious — especially when using the MOVE FIGURE and MOVE 3-D FIGURE programs, which produce sequences of screens to be copied.

The alternative is to copy the screen onto disc as you go, generating files which you can then print at your leisure.

The following program segments can be added to the MOVE FIGURE and MOVE 3-D FIGURE programs to store sequences of pictures automatically as they are produced.

For the MOVE FIGURE program, use this segment:

```
270 PRINT CHR$(4);"BSAVE OUT";STR$(ST);",A16384,L8192"
```

It will cause the program to generate a series of 'binary' files called OUT0, OUT1, OUT2 . . . up to the number of pictures that it has been asked to produce. It will do this without pausing for you to inspect the results, so you can go and have a coffee while it is processing the pictures!

The corresponding segment for the MOVE 3-D FIGURE program is:

```
390 PRINT CHR$(4);"BSAVE OUT";STR$(N9);",A16384,L8192"
```

Adding a printer to your machine will also help you in other ways. By printing a 'listing' of a program, you can reduce development time. If you want to change a program, you can make the change, then RUN the program and look at the listing at the same time. This means that if the program runs into trouble, you can ask for the values of the various variables in the program, and use the listing to find out exactly what the implications are.

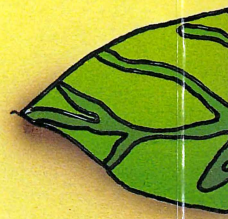
A printer also opens up possibilities for word processing and a host of other applications that require hard copy printing. This book was in fact written on an Apple II. It could be typeset directly from disc without going through another keyboard. The possibilities are worth exploring!

the 3-D animated apple
Phil Cohen

With this book, you will learn how to make full use of the graphics features of one of the most powerful and versatile personal computers around: the Apple II. The Apple's graphics capabilities are extensive and Phil Cohen introduces them to you step by step via the BASIC programming language.

First you will learn how to use your Apple to create simple shapes, and then how to make your shapes move around the Apple's screen. You will learn how to use animation to create video games such as the classic Lunar Lander and how to give your graphics the appearance of three dimensions. Finally, Phil Cohen shows you how to rotate shapes about a point so that you can view them from any perspective — truly 3-D!!

The world of computer graphics is out there waiting for you. Turn to page one and let Phil Cohen introduce you to THE 3-D ANIMATED APPLE.



ISBN 0-13-920224-2

the 3-D animated apple

Phil Cohen

PRENTICE-HALL